

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

Jay McCarthy, Chair
Kevin Seppi
Chris Grant
Eric Mercer
Dan Olsen

Department of Computer Science

Brigham Young University

April 2014

Copyright © 2014 Neil Toronto

All Rights Reserved

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, Measure Theory

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 repurussions 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.

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

Table of Contents

List of Figures	xii
1 Thesis	3
1.1 Introduction	3
1.2 Terms	4
1.3 Proof and Supporting Evidence	5
1.4 Exposition Transition System	6
2 Background	9
2.1 Bayesian Practice	10
2.1.1 Discrete Probability and Joint Distribution Models	10
2.1.2 Probability Densities and Density Models	14
2.1.3 Motivating Measure-Theoretic Models	20
2.2 Functional Programming Theory	21
2.2.1 λ -Calculus	22
2.2.2 Big-Step Operational Semantics	24
2.2.3 Denotational Semantics	29
2.2.4 Categorical Semantics	32
2.2.5 Abstract Interpretation	37
3 Computing in Cantor’s Paradise With λ_{ZFC}	41
3.1 Motivation	41
3.2 Language Tower and Terminology	43

3.3	Metalanguage: First-Order Set Theory	44
3.3.1	The Gateway to Cantor’s Paradise: Infinity	46
3.3.2	Every Set Can Be Sequenced: Well-Ordering	48
3.3.3	Infinity’s Infinity: An Inaccessible Cardinal	48
3.4	λ_{ZFC} ’s Grammar	49
3.4.1	An Infinite Set Rule For Finite BNF Grammars	50
3.4.2	The Grammar of Infinite, Encoded Terms	52
3.5	λ_{ZFC} ’s Big-Step Reduction Semantics	53
3.6	Syntactic Sugar and a Small Set Library	56
3.7	Example: The Reals From the Rationals	58
3.8	Example: Computable Real Limits	60
3.8.1	The Limit Monad	61
3.8.2	The Computable Limit Monad	62
3.9	Related Work	64
3.10	Conclusions	66
4	Using λ_{ZFC}	67
4.1	Computations and Values	67
4.2	Auxiliary Type Systems	68
4.3	Using ZFC Values and Theorems	69
4.4	Internal Equality and External Equivalence	70
4.5	Additional Functions and Syntactic Forms	71
4.6	Extensional Functions	71
5	Countable Models and Implementation	73
5.1	Introduction	73
5.2	The Expression Language	75
5.2.1	Background Theory: Random Variables	75

5.2.2	Interpreting Random Variable Expressions As Computations	76
5.2.3	Implementation in Racket	78
5.3	The Query Language	78
5.3.1	Background Theory: Probability Spaces	79
5.3.2	Background Theory: Queries	79
5.3.3	Interpreting Query Notation	80
5.3.4	Approximating Queries	81
5.3.5	Implementation in Racket	81
5.4	Conditional Queries	82
5.5	The Statement Language	84
5.5.1	Interpreting Common Conditional Theories	84
5.5.2	Interpreting Statements as Monadic Computations	86
5.5.3	Approximating Models and Queries	88
5.5.4	Implementation in Racket	88
5.5.5	Examples	89
5.6	Why Separate Statements and Queries?	91
5.7	Conclusions	92
6	Interlude: Uncountable Outcomes and Recursion	93
7	Preimage Computation Theory: Running Programs Backwards	95
7.1	Introduction	95
7.1.1	Measure-Theoretic Semantics	96
7.1.2	Arrow Solution Overview	97
7.2	Arrows and First-Order Semantics	98
7.2.1	Alternative Arrow Definitions and Laws	98
7.2.2	First-Order Let-Calculus Semantics	102
7.3	The Bottom Arrow	104

7.4	Deriving the Mapping Arrow	106
7.4.1	Composition	108
7.4.2	Pairing	109
7.4.3	Conditional	110
7.4.4	Laziness	111
7.4.5	Correctness	111
7.5	Lazy Preimage Mappings	112
7.5.1	Composition	114
7.5.2	Pairing	115
7.5.3	Disjoint Union	116
7.6	Deriving the Preimage Arrow	117
7.6.1	Composition	119
7.6.2	Pairing	120
7.6.3	Conditional	120
7.6.4	Laziness	121
7.6.5	Correctness	121
7.7	Preimages Under Partial, Probabilistic Functions	122
7.7.1	Motivation	123
7.7.2	Threading and Indexing	123
7.7.3	Applicative, Associative Store Transformer	124
7.7.4	Partial, Probabilistic Programs	125
7.7.5	Correctness	128
7.7.6	Termination	131
7.8	Output Probabilities and Measurability	134
7.9	Approximating Semantics	135
7.9.1	Implementable Lifts	135
7.9.2	Approximate Preimage Mapping Operations	137

7.9.3	Correctness	140
7.9.4	Preimage Refinement Algorithm	142
7.10	Implementations	145
7.11	Conclusions	146
8	Preimage Computation Implementation	147
8.1	Introduction	147
8.2	Abstract Sets and Concrete Values	147
8.2.1	Infinite Binary Trees	153
8.2.2	Disjoint Bottom and Top Unions	157
8.2.3	Testing	159
8.3	Preimages Under Real Functions	163
8.3.1	Invertible Primitives	164
8.3.2	Two-Argument Primitives	166
8.3.3	Primitive Implementation	174
8.3.4	Piecewise Monotone Primitives	181
8.4	Sampling Methods	182
8.4.1	Partitioned Sampling	183
8.4.2	Partitioning Probabilistic Program Domains	187
8.4.3	Approximate Partitions of Probabilistic Program Domains	191
8.4.4	Random Source Sampling	198
8.4.5	Self-Adjusting Probabilistic Search	202
8.5	Conclusions	205
9	Example Programs	207
9.1	Guaranteed Termination	207
9.2	Primitives	211
9.3	Theories With Density Models	215

9.3.1	Normal-Normal	215
9.3.2	Normal-Normals	219
9.3.3	Polynomial Fitting	221
9.3.4	Model Selection	224
9.4	Theories Without Density Models	226
9.4.1	Bounded Measuring Devices	226
9.4.2	Non-Axial Conditions	228
9.4.3	Stochastic Ray Tracing	229
9.5	Current Shortcomings	231
9.6	Conclusions	234
10	Related Work	237
10.1	Implementations	237
10.2	Semantics	238
10.3	Techniques	239
10.4	Somewhat Related Work	240
11	Conclusions and Future Work	241
11.1	Conclusions	241
11.2	Future Work	241
11.2.1	Expressiveness	242
11.2.2	Optimization	243
A	Measurability Theorems	245
A.1	Basic Definitions	245
A.2	Measurable Pure Computations	246
A.2.1	Composition	247
A.2.2	Pairing	249
A.2.3	Conditional	249

A.2.4	Laziness	250
A.3	Measurable Probabilistic Computations	250
A.4	Measurable Projections	253
B	Sampling Theorems	255
B.1	Basic Definitions	255
B.1.1	Measures	255
B.1.2	Integration	257
B.1.3	Differentiation	259
B.1.4	Transition Kernels	263
B.2	Sampling Proofs	264
	References	269

List of Figures

1.1	A transition system for reading this dissertation	7
2.1	Computing probabilities using the standard normal density function	15
2.2	Joint density model plot	16
2.3	Bayes' law for densities, in pictures	19
2.4	Big-step operational semantics example	25
2.5	Implementation of the big-step semantics	27
2.6	Big-step operational semantics with nondeterminism	28
2.7	Denotational semantics and implementation	29
2.8	Denotational semantics with nondeterminism	31
2.9	Categorical semantics with nondeterminism	33
2.10	Abstract semantics with nondeterminism	38
3.1	Definition of λ_{ZFC}^-	49
3.2	Semantic function $\mathcal{F}[\![\cdot]\!]$	50
3.3	λ_{ZFC} 's grammar	53
3.4	λ_{ZFC} 's semantics	54
5.1	Random variable expression semantics	77
5.2	Implementation of $\mathcal{R}[\![\cdot]\!]$	78
5.3	Implementation of finite approximation and distribution queries	82
5.4	State monad functions for queries and statements	86
5.5	Theory extension and query semantic functions	88

7.1	First-order semantics	103
7.2	Bottom arrow definitions	105
7.3	Additional mapping operations	107
7.4	Mapping arrow definitions	107
7.5	Lazy preimage mappings	113
7.6	Comparison of arrows used as target categories	118
7.7	Preimage arrow definitions	119
7.8	Associative store arrow transformer	125
7.9	A random source $\omega \in \Omega$	126
7.10	Specific preimage arrow lifts	135
7.11	A finite model of a rectangular subset of Ω	137
7.12	Implementable, approximating arrows	140
7.13	Preimage refinement algorithm	144
8.1	Haskell typeclass for rectangular sets	148
8.2	Haskell implementation of sets of pairs	149
8.3	Typed Racket implementation of rectangular sets	150
8.4	Typed Racket implementation of intervals	151
8.5	Typed Racket representation of $\text{Rect } \Omega$	155
8.6	Typed Racket representation of values $\omega \in \Omega$	156
8.7	Computing the preimage of $[2, 7]$	165
8.8	Computing an approximate preimage of $[0, 1/2]$	169
8.9	Multiplication on $\mathbb{R} \times \mathbb{R}$	170
8.10	Computing images and preimages under real functions	178
8.11	Computing images and preimages under two-dimensional real functions	180
8.12	Preimage refinement sampling	182
8.13	Sampling uniformly in a partitioned unit square	186
8.14	Branch index collecting semantics	192

8.15	Final indexes arrow definitions	195
8.16	Preimage refinement sampling	196
8.18	Self-adjusting, probabilistic tree search	203
8.19	Self-adjusting, probabilistic tree search algorithm	203
9.1	Sampling from preimages under multiplication and division	212
9.2	Samples from Dr. Bayes and a density estimate	217
9.3	Nested rectangular conditions	220
9.4	Inference with ϵ_i of differing magnitudes	221
9.5	Bayesian analysis of Dr. Bayes's running time, using Dr. Bayes	222
9.6	Bayesian theory selection in Dr. Bayes	225
9.7	Bayesian inference with a bounded measuring device	227
9.8	Circular probabilistic conditions	229
9.9	Stochastic ray tracing in Dr. Bayes	230
9.10	The dependency problem	233

“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 λ -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 λ -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 λ -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 λ_{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 program-

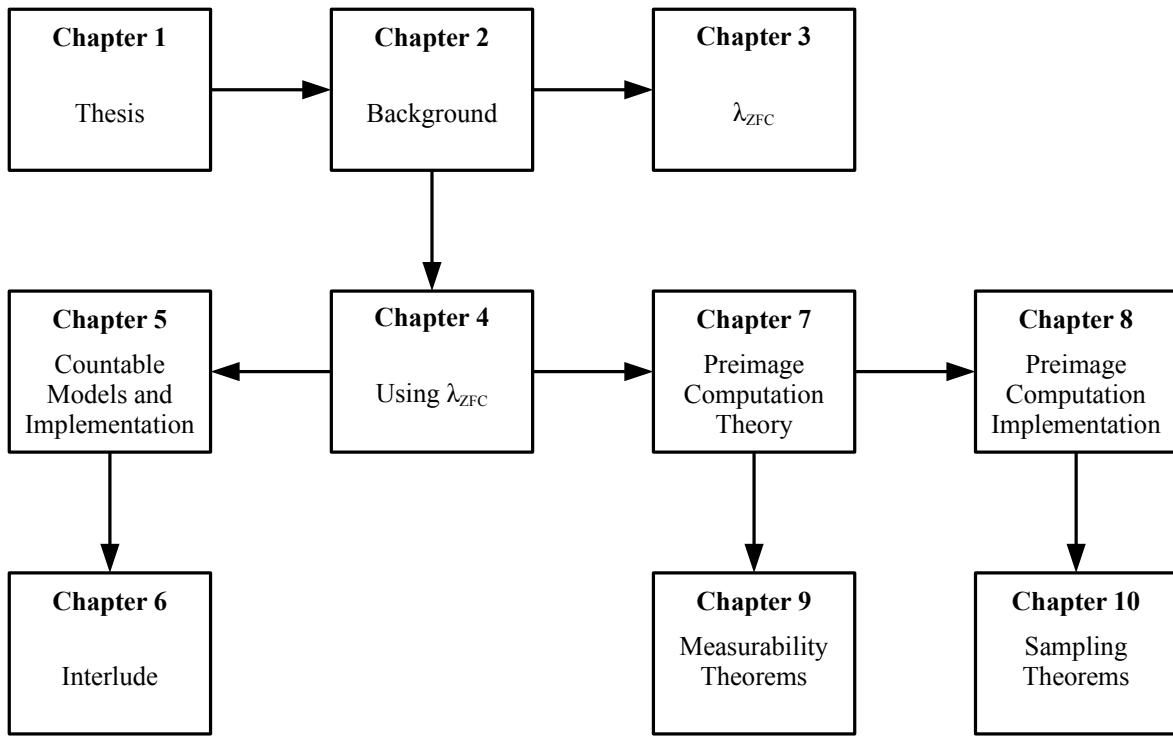


Figure 1.1: A transition system showing possible paths through this dissertation.

ming theory, and motivates using measure theory.

The two semantics mentioned in the preceding section transform programs into λ -calculus terms. This target language has three requirements that are unusual for a λ -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 λ -calculus did not exist. Chapter 3 defines one, λ_{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 λ_{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.

Appendix A 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. Appendix B 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 $p_{X,Y} : \{h, t\} \times \{h, t\} \rightarrow [0, 1]$ quantify the likelihood of every possible combination of observable outcomes by defining

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

(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**:

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

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 :

$$\Pr[X = x] = \sum_{y \in \{h,t\}} \Pr[X = x, Y = y] \quad (2.3)$$

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

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

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

$$\Pr[Y = y | X = x] = \frac{\Pr[X = x, Y = y]}{\Pr[X = x]} \quad (2.5)$$

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:

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

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 preceding 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

¹It 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

$$\Pr[X = x | Y = y] = \frac{\Pr[X = x] \cdot \Pr[Y = y | X = x]}{\Pr[Y = y]} \quad (2.10)$$

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

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

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:

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

²For 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*.

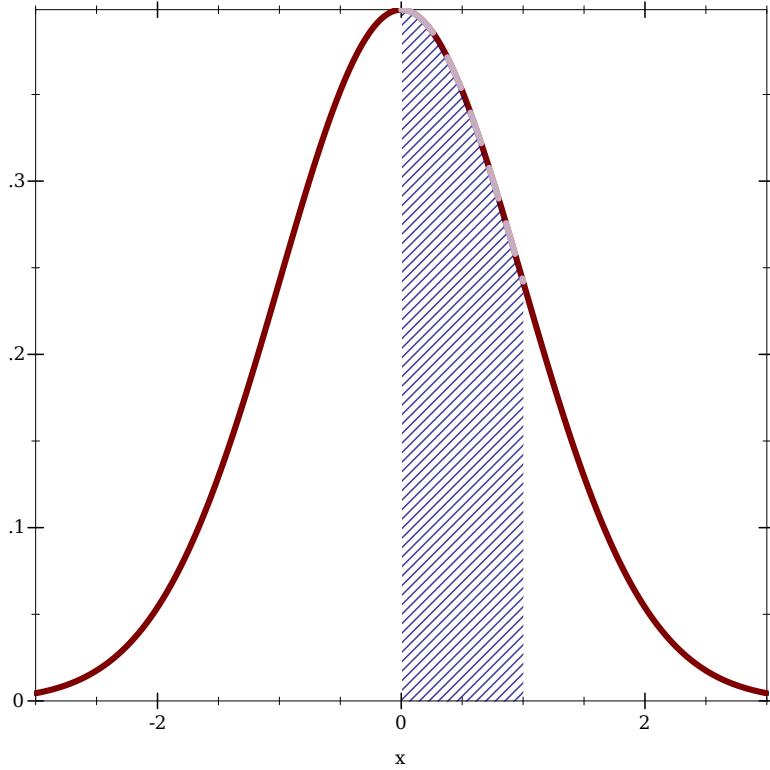


Figure 2.1: Integrating under the standard normal density to compute $\Pr[X \in (0, 1)] \approx 0.34$.

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

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

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 $f_X : \mathbb{R} \rightarrow [0, +\infty)$,

$$\Pr[X = x] = \int_x^x f_X(x) dx = (f_X(x) - 0) \cdot (x - x) = f_X(x) \cdot 0 = 0 \quad (2.15)$$

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

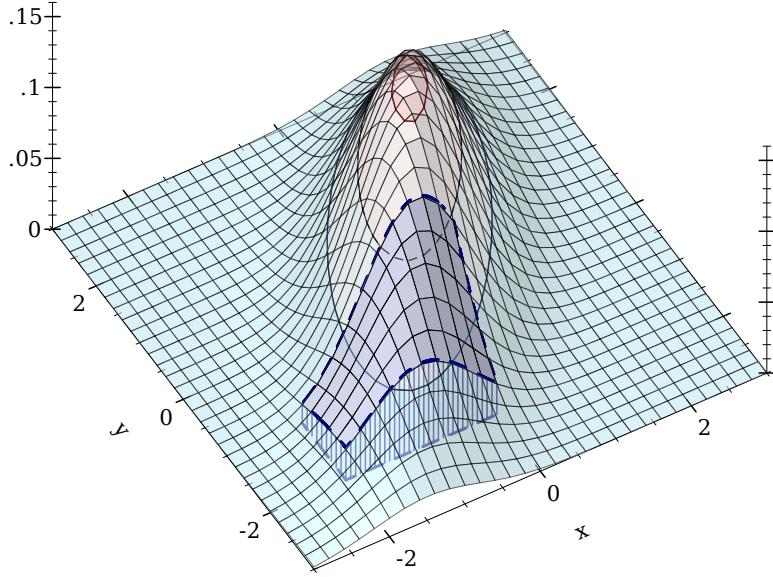


Figure 2.2: The joint density model $f_{X,Y}$ constructed from the density f_X and the conditional density $f_{Y|X}$. Integrating under $f_{X,Y}$ on the set $(-2, 0) \times (-2, -1)$ computes $\Pr[X \in (-2, 0), Y \in (-2, -1)]$.

it on a *mean* μ and a *standard deviation* σ :

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

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

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

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

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

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,

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

$$= \frac{f_X(x) \cdot f_{Y|X}(y | x)}{\int_{-\infty}^{+\infty} f_X(x) \cdot f_{Y|X}(y | x) dx}$$

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

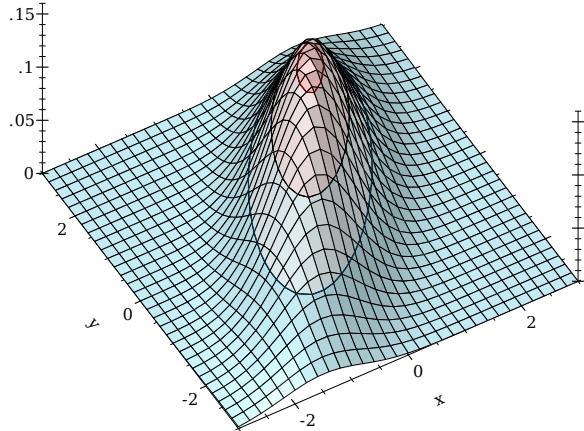
$$\begin{aligned} f_{X|Y}(x | 2) &= \frac{f_X(x) \cdot f_{Y|X}(2 | x)}{\int_{-\infty}^{+\infty} f_X(x) \cdot f_{Y|X}(2 | x) dx} \\ &= \frac{f_N(x | 0, 1) \cdot f_N(2 | x, 1)}{\int_{-\infty}^{+\infty} f_N(x | 0, 1) \cdot f_N(2 | x, 1) dx} \\ &\quad \dots \\ &= f_N(x | 1, \sqrt{\frac{1}{2}}) \end{aligned}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

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

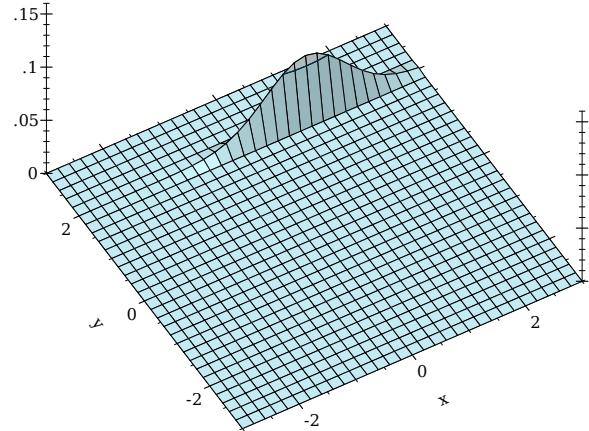
$$\Pr[X \in (a, b) | Y = 2] = \int_a^b f_N(x | 1, \sqrt{\frac{1}{2}}) dx \tag{2.23}$$

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, 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).

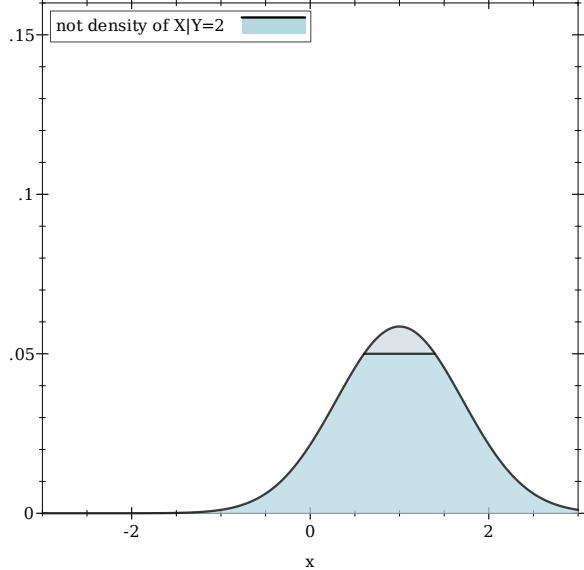
Using Bayes' law for densities is not only often technically challenging, but in general there are no closed-form solutions. In such cases, practitioners turn to Monte Carlo integration,



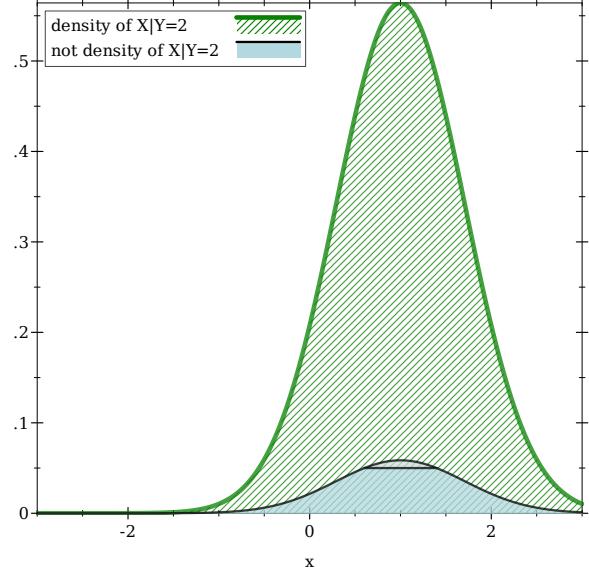
(a) The original joint density model.



(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.

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:

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

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

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

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 preceding examples, on a part of the distribution's domain that has length or area 0 (i.e. the single outcome 100 or the set $\{x \in \mathbb{R}, y \in \mathbb{R} \mid \sqrt{x^2 + y^2} = z\}$), 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 $\Pr[X \in (a, b), Y \in (c, d) \mid \sqrt{X^2 + Y^2} = 1]$. 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 λ -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 λ -Calculus

The following grammar defines a set of variable names X and a language E (a set of terms) called the **pure λ -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 λ -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 λ -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

$$hypot = \lambda x. \lambda y. \sqrt{x^2 + y^2} \quad (2.28)$$

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

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

The two equivalences at the end of (2.29) are called **β -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 **indexes**. One form of the pure λ -calculus with De Bruijn indexes is

$$\begin{aligned} e ::= & \mathsf{env}\ n \mid e\ e \mid \lambda. e \\ n ::= & 0 \mid 1 \mid 2 \mid \dots \end{aligned} \quad (2.30)$$

where a “variable” term $\mathsf{env}\ 0$ refers to the innermost lambda’s argument.

Suppose we define *hypot* as a term in a λ -calculus with De Bruijn indexes, extended

³Typically pronounced “deh brOIN,” and named after Dutch mathematician Nicolaas de Bruijn.

with real numbers:

$$hypot = \lambda. \lambda. \sqrt{(\text{env } 1)^2 + (\text{env } 0)^2} \quad (2.31)$$

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

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

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:

$$zero = \lambda x. 0 \quad (2.33)$$

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:

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

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 λ -calculus a **semantics**: a precise mathematical definition of the meaning of its terms. To specify evaluation order and other operational aspects

$\begin{array}{l} e ::= v \mid \text{add } e \ e \\ v ::= 0 \mid 1 \mid 2 \mid \dots \end{array}$	$\frac{}{v \Downarrow v} (\text{val}) \quad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\text{add } e_1 \ e_2) \Downarrow (v_1 + v_2)} (\text{add})$
(a) A grammar to define sets E and V	(b) Reduction rules to define $\Downarrow \subseteq E \times V$

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

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

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

A small-step semantics would typically “stop” after a function application. If “ \Rightarrow ” is a small-step reduction relation, then $(inc \ 4) \Rightarrow (4 + 1)$ 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 $(inc \ 4) \Downarrow (4 + 1)$ because by any reasonable definition, the term $4 + 1$ is an *expression* but not a *value*. We should expect, however, that $(inc \ 4) \Downarrow 5$ 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 dissertation 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 λ -calculus: its terms simply represent adding concrete numbers. The relation “ \Downarrow ” is defined by **reduction rules**

in the form

$$\frac{\text{premise}_1 \quad \text{premise}_2 \quad \dots}{\text{conclusion}} \quad (\text{name}) \quad (2.36)$$

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 $\text{add } e_1 e_2$ 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 $\text{add } (\text{add } 4 5) 90$. We start by writing it as a conclusion without premises:

$$\overline{(\text{add } (\text{add } 4 5) 90) \Downarrow v_1} \quad (2.37)$$

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

$$\frac{\overline{(\text{add } 4 5) \Downarrow v_2} \quad \overline{90 \Downarrow v_3}}{(\text{add } (\text{add } 4 5) 90) \Downarrow v_1} \quad (2.38)$$

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 $(\text{add } 4 5)$:

$$\frac{\overline{4 \Downarrow v_4} \quad \overline{5 \Downarrow v_5}}{\overline{(\text{add } 4 5) \Downarrow v_2} \quad \overline{90 \Downarrow v_3}} \quad \overline{(\text{add } (\text{add } 4 5) 90) \Downarrow v_1} \quad (2.39)$$

It is easy to find values of v_3 , v_4 and v_5 that make the leaf premises true, so we substitute

```

(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.

them and recursively fill in the conclusions:

$$\frac{\overline{4 \Downarrow 4} \quad \overline{5 \Downarrow 5}}{(\text{add } 4 5) \Downarrow v_2} \quad \frac{\overline{90 \Downarrow v_3}}{} \Rightarrow \frac{\overline{4 \Downarrow 4} \quad \overline{5 \Downarrow 5}}{(\text{add } 4 5) \Downarrow 9} \quad \frac{\overline{90 \Downarrow 90}}{} \Rightarrow \frac{\overline{4 \Downarrow 4} \quad \overline{5 \Downarrow 5}}{(\text{add } 4 5) \Downarrow 9} \quad \frac{\overline{90 \Downarrow 90}}{} \Rightarrow \frac{\overline{(\text{add } 4 5) \Downarrow 9} \quad \overline{90 \Downarrow 90}}{(\text{add } (\text{add } 4 5) 90) \Downarrow v_1} \Rightarrow \frac{\overline{(\text{add } 4 5) \Downarrow 9} \quad \overline{90 \Downarrow 90}}{(\text{add } (\text{add } 4 5) 90) \Downarrow 99} \quad (2.40)$$

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

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 $\Downarrow : E \rightarrow V$. 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

$e ::= v \mid \text{add } e \ e \mid \text{choose } e \ e$ $v ::= 0 \mid 1 \mid 2 \mid \dots$	$\frac{}{v \Downarrow v} (\text{val}) \quad \frac{e_1 \Downarrow v_1 \ e_2 \Downarrow v_2}{(\text{add } e_1 \ e_2) \Downarrow (v_1 + v_2)} (\text{add})$ $\frac{e_1 \Downarrow v_1}{(\text{choose } e_1 \ e_2) \Downarrow v_1} (\text{left}) \quad \frac{e_2 \Downarrow v_2}{(\text{choose } e_1 \ e_2) \Downarrow v_2} (\text{right})$
(a) A grammar to define sets E and V	(b) Reduction rules to define $\Downarrow \subseteq E \times V$

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

in (2.37) through (2.40). As an example of use, at DrRacket’s Read-Eval-Print Loop (REPL), 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 `add (choose 4 5) 90`. We start as before, by writing it as a conclusion without premises:

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

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

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

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

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

$\llbracket \cdot \rrbracket : E \rightarrow \mathbb{N}$ $\llbracket v \rrbracket = v$ $\llbracket \text{add } e_1 e_2 \rrbracket = \llbracket e_1 \rrbracket + \llbracket e_2 \rrbracket$	<pre>(define-syntax compile (syntax-rules (add) [(_ (add e1 e2)) (+ (compile e1) (compile e2))]</pre>
(a) A semantic function for the addition language	(b) An implementation of the semantic function as a syntax transformer

Figure 2.7: A denotational semantics and its implementation.

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 $(\text{add} (\text{choose } 4 5) 90) \Downarrow 94$ and $(\text{add} (\text{choose } 4 5) 90) \Downarrow 95$ 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 $\text{choose } 4 5$, 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 $\Downarrow : E \rightarrow \mathcal{P} V$, 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 mathematics).

Figure 2.7a defines a denotational semantics for the addition language without `choose` by

defining a semantic function $\llbracket \cdot \rrbracket : E \rightarrow \mathbb{N}$. 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

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

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 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)
```

$$\begin{aligned}\llbracket \cdot \rrbracket : E \rightarrow \mathcal{P} \mathbb{N} \\ \llbracket v \rrbracket &= \{v\} \\ \llbracket \text{add } e_1 \ e_2 \rrbracket &= \{v_1 + v_2 \mid v_1 \in \llbracket e_1 \rrbracket, v_2 \in \llbracket e_2 \rrbracket\} \\ \llbracket \text{choose } e_1 \ e_2 \rrbracket &= \llbracket e_1 \rrbracket \cup \llbracket e_2 \rrbracket\end{aligned}$$

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

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

$$\begin{aligned}\llbracket \text{add (choose 4 5) 90} \rrbracket &= \{v_1 + v_2 \mid v_1 \in \llbracket \text{choose 4 5} \rrbracket, v_2 \in \llbracket 90 \rrbracket\} \\ &= \{v_1 + v_2 \mid v_1 \in (\llbracket 4 \rrbracket \cup \llbracket 5 \rrbracket), v_2 \in \llbracket 90 \rrbracket\} \\ &= \{v_1 + v_2 \mid v_1 \in (\{4\} \cup \{5\}), v_2 \in \{90\}\} \\ &= \{v_1 + v_2 \mid v_1 \in \{4, 5\}, v_2 \in \{90\}\} \\ &= \{4 + 90, 5 + 90\} \\ &= \{94, 95\}\end{aligned}\tag{2.45}$$

We know that under “ \Downarrow ,” `add (choose 4 5) 90` reduces to both 94 and 95, so it appears $\llbracket \cdot \rrbracket$ 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 $\llbracket \cdot \rrbracket$ ’s compositionality in a proof by induction on the structure of e .

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

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 $v \in \llbracket e \rrbracket = \{e\} = \{v\}$ by definition of $\llbracket \cdot \rrbracket$, and $e \Downarrow v$ by the (val) rule. Similarly, if $e \neq v$, then $v \notin \llbracket e \rrbracket$, and not $e \Downarrow v$.

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

Suppose $v \in \llbracket \text{add } e_1 \ e_2 \rrbracket$. By definition of $\llbracket \cdot \rrbracket$, there exist $v_1 \in \llbracket e_1 \rrbracket$, $v_2 \in \llbracket e_2 \rrbracket$ 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, $(\text{add } e_1 e_2) \Downarrow v$.

Conversely, if $(\text{add } e_1 e_2) \Downarrow v$, 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, $v_1 \in \llbracket e_1 \rrbracket$ and $v_2 \in \llbracket e_2 \rrbracket$. By definition of $\llbracket \cdot \rrbracket$, $v \in \llbracket \text{add } e_1 e_2 \rrbracket$.

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

Now that we know $\llbracket \cdot \rrbracket$ 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 $\llbracket \cdot \rrbracket$, 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 probabilistic 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

⁴The word “category” comes from category theory, an alternative axiomatization of mathematics. Fortunately, little knowledge of category theory is necessary to define or understand categorical semantics.

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

$$\begin{aligned}\llbracket \cdot \rrbracket_a : E \rightarrow M_a \mathbb{N} \\ \llbracket v \rrbracket_a &= \text{return}_a v \\ \llbracket \text{add } e_1 \ e_2 \rrbracket_a &= \text{bind}_a \llbracket e_1 \rrbracket_a (\lambda v_1. \text{bind}_a \llbracket e_2 \rrbracket_a (\lambda v_2. \text{return}_a (v_1 + v_2))) \\ \llbracket \text{choose } e_1 \ e_2 \rrbracket_a &= \text{merge}_a \llbracket e_1 \rrbracket_a \llbracket e_2 \rrbracket_a\end{aligned}$$

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

defined by these two combinators:

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

Evidently, from a semantic function $\llbracket \cdot \rrbracket_a$ parameterized on a monad a we should expect $\llbracket v \rrbracket_{set} = \text{return}_{set} v \equiv \{v\}$. How to use bind_{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 $\llbracket \text{add } e_1 \ e_2 \rrbracket$ into an indexed union (as in bind_{set}) makes its use clearer:

$$\begin{aligned}\llbracket \text{add } e_1 \ e_2 \rrbracket &= \{v_1 + v_2 \mid v_1 \in \llbracket e_1 \rrbracket, v_2 \in \llbracket e_2 \rrbracket\} \\ &= \bigcup_{v_1 \in \llbracket e_1 \rrbracket} \bigcup_{v_2 \in \llbracket e_2 \rrbracket} \{v_1 + v_2\} \\ &\equiv \bigcup_{v_1 \in \llbracket e_1 \rrbracket} \bigcup_{v_2 \in \llbracket e_2 \rrbracket} \text{return}_{set} (v_1 + v_2) \\ &\equiv \bigcup_{v_1 \in \llbracket e_1 \rrbracket} \text{bind}_{set} \llbracket e_2 \rrbracket (\lambda v_2. \text{return}_{set} (v_1 + v_2)) \\ &\equiv \text{bind}_{set} \llbracket e_1 \rrbracket (\lambda v_1. \text{bind}_{set} \llbracket e_2 \rrbracket (\lambda v_2. \text{return}_{set} (v_1 + v_2)))\end{aligned}\tag{2.47}$$

Thus, we expect $\llbracket \text{add } e_1 \ e_2 \rrbracket_{set} = \text{bind}_{set} \llbracket e_1 \rrbracket_{set} (\lambda v_1. \text{bind}_{set} \llbracket e_2 \rrbracket_{set} (\lambda v_2. \text{return}_{set} (v_1 + v_2)))$.

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

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

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

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 $\llbracket \cdot \rrbracket_{set} : E \rightarrow M_{set} \mathbb{N}$ is equivalent to $\llbracket \cdot \rrbracket : E \rightarrow \mathcal{P} \mathbb{N}$ 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

$$\begin{aligned} return_{list} v &= [v] \\ bind_{list} vs f &= concat (map f vs) \\ merge_{list} vs_1 vs_2 &= append vs_1 vs_2 \end{aligned} \tag{2.49}$$

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 *concat* : $[[X]] \rightarrow [X]$ appends the elements in a list of lists. Now $\llbracket \cdot \rrbracket_{list} : E \rightarrow [\mathbb{N}]$ models nondeterminism with lists of numbers instead of sets of numbers. For example, the meaning of **choose** 4 5 as a list of nondeterministic choices is

$$\begin{aligned} \llbracket \text{choose } 4 \ 5 \rrbracket_{list} &= merge_{list} \llbracket 4 \rrbracket_{list} \llbracket 5 \rrbracket_{list} \\ &= merge_{list} (return_{list} 4) (return_{list} 5) \\ &\equiv merge_{list} [4] [5] \\ &\equiv append [4] [5] \\ &\equiv [4, 5] \end{aligned} \tag{2.50}$$

The meaning of **add** (**choose** 4 5) (**choose** 4 5) is thus

$$\begin{aligned} \llbracket \text{add } (\text{choose } 4 \ 5) \ (\text{choose } 4 \ 5) \rrbracket_{list} &\equiv bind_{list} [4, 5] (\lambda v_1. bind_{list} [4, 5] (\lambda v_2. return_{list} (v_1 + v_2))) \\ &\equiv concat (map (\lambda v_1. bind_{list} [4, 5] (\lambda v_2. return_{list} (v_1 + v_2))) [4, 5]) \end{aligned}$$

$$\begin{aligned}
&\equiv concat [bind_{list} [4, 5] (\lambda v_2. return_{list} (4 + v_2)), \\
&\quad bind_{list} [4, 5] (\lambda v_2. return_{list} (5 + v_2))] \\
&\equiv concat [concat (map (\lambda v_2. return_{list} (4 + v_2)) [4, 5]), \\
&\quad concat (map (\lambda v_2. return_{list} (5 + v_2)) [4, 5])] \\
&\equiv concat [concat [return_{list} (4 + 4), return_{list} (4 + 5)], \\
&\quad concat [return_{list} (5 + 4), return_{list} (5 + 5)]] \\
&\equiv concat [concat [[8], [9]], concat [[9], [10]]] \\
&\equiv concat [[8, 9], [9, 10]] \\
&\equiv [8, 9, 9, 10]
\end{aligned} \tag{2.51}$$

In contrast, $\llbracket \text{add (choose } 4 \ 5 \text{) (choose } 4 \ 5 \text{)} \rrbracket_{set} \equiv \{8, 9, 10\}$.

The semantic function $\llbracket \cdot \rrbracket_a$ can be parameterized not just on the set and list monads, but any monad a for which $merge_a$ 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 λ -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 $\llbracket \cdot \rrbracket_a$ targets a λ -calculus.

Targeting a λ -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 λ -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:

$$\begin{aligned} \llbracket \text{add } e_1 \ e_2 \rrbracket &= \{v_1 + v_2 \mid v_1 \in \llbracket e_1 \rrbracket, v_2 \in \llbracket e_2 \rrbracket\} \\ \llbracket \text{add } e_1 \ e_2 \rrbracket_a &= \text{bind}_a \ \llbracket e_1 \rrbracket_a \ (\lambda v_1. \text{bind}_a \ \llbracket e_2 \rrbracket_a \ (\lambda v_2. \text{return}_a (v_1 + v_2))) \end{aligned} \tag{2.52}$$

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

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

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 λ -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, 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**.

$$\begin{aligned}\mathcal{L}[\cdot] : E &\rightarrow \mathbb{N} \\ \mathcal{L}[v] &= 1 \\ \mathcal{L}[\text{add } e_1 e_2] &= \mathcal{L}[e_1] \cdot \mathcal{L}[e_2] \\ \mathcal{L}[\text{choose } e_1 e_2] &= \mathcal{L}[e_1] + \mathcal{L}[e_2]\end{aligned}$$

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

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 `add (choose 4 5) (choose 4 5)` is

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

Indeed, $[\![\text{add } (\text{choose } 4 5) (\text{choose } 4 5)]\!]_{\text{set}} \equiv \{8, 9, 10\}$, 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 $e \in E$, $|\llbracket e \rrbracket_{set}| \leq \mathcal{L}[e]$.*

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 $\llbracket \cdot \rrbracket_{set}$, and we will never allocate too little.

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

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

Proof. By structural induction on e . □

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 λ -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 λ -calculus.

Chapter 3

Computing in Cantor’s Paradise With λ_{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 λ -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 λ -calculus that contained infinite sets and operations on them. We could interpret DSL terms exactly as uncomputable programs in this λ -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 λ -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 λ -calculus, λ_{ZFC} , and a call-by-value, big-step reduction semantics. To

show that it is expressive enough, we code up the real numbers, arithmetic and limits, following standard analysis. To show that it simplifies language design, we define the uncomputable limit monad in λ_{ZFC} , and derive a computable, directly implementable replacement monad by applying standard topological theorems. When certain proof obligations are met, the 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

λ_{ZFC} ’s metalanguage is **first-order set theory**: first-order logic with equality extended with ZFC, or the Zermelo Fraenkel axioms and Choice (equivalently well-ordering). We also assume the existence of an inaccessible cardinal. Section 3.3 reviews the axioms, from which we will derive λ_{ZFC} ’s primitives.

To help ensure λ_{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} .

λ_{ZFC} ’s semantics reduces terms to terms; e.g. $\langle t_{\mathcal{P}}, \mathbb{R} \rangle$ reduces to the actual powerset of \mathbb{R} . Thus, λ_{ZFC} contains infinite terms. Infinitary languages are useful and definable: the infinitary λ -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 λ_{ZFC}^- of finite terms and a function $\mathcal{F}[\cdot]$ from λ_{ZFC}^- to λ_{ZFC} . We can then write $\mathcal{P} \mathbb{R}$, meaning $\mathcal{F}[\mathcal{P} \mathbb{R}] = \langle t_{\mathcal{P}}, \mathbb{R} \rangle$.

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

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

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 ϕ and object-valued terms x . Set theory allows only sets as objects, and quantifiers like “ \forall ” may range only over sets.

We define predicates and functions using “ $::=$ ”; 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 “ \emptyset ” and “ \in ”, and the **empty set axiom** $\forall x. x \notin \emptyset$.

We use “ $::\equiv$ ” to define syntax; e.g. $\forall x \in A. P(x) ::\equiv \forall x. (x \in A \Rightarrow P(x))$, where predicate application $P(x)$ represents a formula that may depend on x . We allow recursion in meta-metalanguage definitions if substitution terminates, so $\forall x_1 x_2 \dots x_n. \phi ::\equiv \forall x_1. \forall x_2 \dots x_n. \phi$ 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 $\forall A B. (A \subseteq B \wedge B \subseteq A \Rightarrow A = B)$. \square

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

Axiom 2 (foundation). Define $A \not\supseteq B := \forall x. (x \in A \Rightarrow x \notin B)$ (“ A and B are disjoint”) and assume $\forall A. (A = \emptyset) \vee \exists x \in A. x \not\in A$. \square

Foundation implies that the following nondeterministic procedure always terminates: If input $A = \emptyset$, return A ; otherwise restart with any $A' \in A$. Thus, sets are roots of trees in which every upward path is unbounded but finite. Foundation is analogous to “all data constructors are strict.”

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

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

Axiom 4 (union). Add “ \bigcup ” (“big” union) and assume arbitrary unions of sets of sets exist; i.e. $\forall A x. (x \in \bigcup A \iff \exists y. x \in y \wedge y \in A)$. \square

For example, after $\{x, \dots\}$ 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. \square

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 $\{F(x) \mid x \in A\} := \{y \mid x \in A \wedge y = F(x)\}$, analogous to **map** $\text{F } A$, 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**

$$\iota y. Q(y) := \bigcup\{y \mid x \in \mathcal{P}(\emptyset) \wedge Q(y)\} \tag{3.1}$$

finds “the y such that $Q(y)$.”

From the six axioms so far, we can define $A \cup B$ (binary union), $\{x, \dots\}$ (literal finite sets), $\langle x, y, z, \dots \rangle$ (ordered pairs and lists), $\{x \in A \mid Q(x)\}$ (bounded selection), $A \setminus B$ (relative complement), $\cap A$ (“big” intersection), $\bigcup_{x \in A} F(x)$ (indexed union), $A \times B$ (cartesian product), and $A \rightarrow B$ (total function spaces). For details, we recommend Paulson’s remarkably lucid development in HOL [42].

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 $z ::= \emptyset \mid \langle \emptyset, z \rangle$. 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 “ \mid ” with “ \cup ”, the terminal \emptyset with $\{\emptyset\}$, and the rule $\langle \emptyset, z \rangle$ with functional replacement:

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

We could define $Z(0) := \emptyset$, then $Z(1) := F_z(Z(0)) = \{\emptyset, \langle \emptyset, \emptyset \rangle\}$, then $Z(2) = F_z(Z(1)) = \{\emptyset, \langle \emptyset, \emptyset \rangle, \langle \emptyset, \emptyset, \emptyset \rangle\}$, 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 . \diamond

We follow Von Neumann, defining $0 := \emptyset$ 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-Tarski theorem (suitably restricted [43]),

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

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 $z ::= \emptyset \mid \langle \emptyset, z \rangle$ recursively:

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

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 \emptyset , and that it is the least fixpoint of F_z . \diamond

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 :

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

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

Defining more limit ordinals allows iterating \mathcal{P} further. It is not hard to build $\omega + \omega$, ω^2

and ω^ω as least fixpoints. The **Von Neumann hierarchy** generalizes (3.5):

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

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

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

3.3.2 Every Set Can Be Sequenced: Well-Ordering

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

Axiom 7 (well-ordering). Suppose the predicate Ord identifies ordinals and $B \leftrightarrow A$ is the set of all bijective mappings from B to A . Assume $\forall A. \exists \alpha f. Ord(\alpha) \wedge f \in \alpha \leftrightarrow A$; i.e. every set can be well-ordered. \square

Because the bijective sequence f is not unique, a well-ordering primitive could make λ_{ZFC} 's semantics nondeterministic. Fortunately, the existence of a cardinality operator is equivalent to well-ordering [52], so we will give λ_{ZFC} a cardinality primitive.

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

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.

$$\begin{aligned}
e &::= n \mid v \mid e \, e \mid \text{if } e \, e \, e \mid e \in e \mid \cup \, e \mid \text{take } e \mid \mathcal{P} \, e \mid \text{image } e \, e \mid \text{card } e \\
v &::= \text{false} \mid \text{true} \mid \lambda. \, e \mid \emptyset \mid \omega \\
n &::= 0 \mid 1 \mid 2 \mid \dots
\end{aligned}$$

Figure 3.1: The definition of λ_{ZFC}^- , which represents countably many λ_{ZFC} terms.

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

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

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

Inaccessible cardinals are not usually assumed but are widely believed consistent. Set theorists regard them as no more dangerous than ω . Interpreting category theory with small and large categories, second-order set theory, or CIC in first-order set theory requires at least one inaccessible cardinal [6, 53, 56].

Constructing a set $A \notin \mathcal{V}(\kappa)$ requires assuming κ or an equivalent, so $\mathcal{V}(\kappa)$ easily contains most mathematics. In fact, most can be modeled well within $\mathcal{V}(2^\omega)$; e.g. the model of \mathbb{R} we define in Section 3.7 is in $\mathcal{V}(\omega + 11)$. Besides, if λ_{ZFC} needed to contain large cardinals, we could always assume even larger ones.

3.4 λ_{ZFC} 's Grammar

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

Figure 3.1 shows λ_{ZFC}^- 's grammar. Expressions e are typical: variables, values, application, if, and domain-specific primitives, for membership, union, extraction (take), powerset,

Distinct $t_{\text{var}}, t_{\text{app}}, t_{\text{if}}, t_{\in}, t_{\cup}, t_{\text{take}}, t_{\mathcal{P}}, t_{\text{image}}, t_{\text{card}}, t_{\text{set}}, t_{\text{atom}}, t_{\lambda}, t_{\text{false}}, t_{\text{true}}$	
$\mathcal{F}[\![n]\!] := \langle t_{\text{var}}, n \rangle$	$\mathcal{F}[\!\![\emptyset]\!] := \text{set}(\emptyset)$
$\mathcal{F}[\![e_f e_x]\!] := \langle t_{\text{app}}, \mathcal{F}[\![e_f]\!], \mathcal{F}[\![e_x]\!] \rangle$	$\mathcal{F}[\!\![\text{false}]\!] := a_{\text{false}}$
$\mathcal{F}[\![e_x \in e_A]\!] := \langle t_{\in}, \mathcal{F}[\![e_x]\!], \mathcal{F}[\![e_A]\!] \rangle$	$a_{\text{false}} := \langle t_{\text{atom}}, t_{\text{false}} \rangle$
...	$\mathcal{F}[\!\![\text{true}]\!] := a_{\text{true}}$
	$a_{\text{true}} := \langle t_{\text{atom}}, t_{\text{true}} \rangle$
	$\text{set}(A) = \langle t_{\text{set}}, \{ \text{set}(x) \mid x \in A \} \rangle$

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

functional replacement (`image`), and cardinality. Values v are also typical: booleans and lambdas, and the domain-specific constants \emptyset and ω .

In set theory, $\bigcup \{A\} = A$ holds for all A , so \bigcup can extract the element from a singleton. In λ_{ZFC} , the encoding of $\bigcup \{A\}$ reduces to A only if A is an encoded set. Therefore, the primitives must include `take`, which extracts A from $\{A\}$. In particular, extracting a lambda from an ordered pair requires `take`.

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

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

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

3.4.1 An Infinite Set Rule For Finite BNF Grammars

There is no sensible reduction relation for λ_{ZFC}^- . (For example, $\mathcal{P} \emptyset$ cannot correctly reduce to a value because no value in λ_{ZFC}^- corresponds to $\{\emptyset\}$.) The easiest way to ensure a reduction

relation exists for λ_{ZFC} is to include encodings of all the sets in $\mathcal{V}(\kappa)$ as values.

To define λ_{ZFC} 's terms, we first extend BNF with a set rule: $\{y^{*\alpha}\}$, where α is a cardinal number. Roughly, it means sets comprised of no more than α 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

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

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

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 $h ::= \{\} \mid \{h\} \mid \{h, h\} \mid \{h, h, h\} \mid \dots$.

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. \diamond

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 $\mathcal{V}(\kappa)$. \diamond

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:*

$$\begin{aligned} V(0) &= \emptyset \\ V(s(\alpha)) &= F(V(\alpha)) \\ V(\beta) &= \bigcup_{\alpha \in \beta} V(\alpha), \text{ limit ordinal } \beta \end{aligned} \quad (3.8)$$

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

Proof. By induction: successor case by monotonicity; limit by 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 \emptyset .

From here on, we write F^α instead of $V(\alpha)$ to mean α 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^γ , where γ is a regular cardinal not less than any α .*

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

Example 3.7 (finite sets). Because ω is regular, by Theorem 3.6, F_h ’s least fixpoint is F_h^ω . Further, $F_h(H) = \mathcal{P}(H)$ for all hereditarily finite H , and $\mathcal{V}(\omega)$ is closed under \mathcal{P} , so $F_h^\omega = \mathcal{V}(\omega)$, the set of all hereditarily finite sets. \diamond

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. \diamond

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. \diamond

3.4.2 The Grammar of Infinite, Encoded Terms

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

The language of n is $N := \{\langle t_{\text{var}}, i \rangle \mid i \in \omega\}$. The rules for e and v are mutually recursive.

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

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

Interpreted, but leaving out some of e 's rules, they are

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

To use Theorem 3.6, we need to iterate a single function. Note that the language pair $\langle E, V \rangle = \langle \{e, \dots\}, \{v, \dots\} \rangle$ is isomorphic to the single set of tagged terms $EV = \{\langle 0, e \rangle, \dots, \langle 1, v \rangle, \dots\}$. 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}^κ , so we define E and V by $E \sqcup V = F_{ev}^\kappa$.

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

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

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

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

(a) Standard call-by-value reduction rules

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

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

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

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

(b) λ_{ZFC} -specific rules

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

resulting cardinal number. The rest are

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

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

We use $e[n \setminus v]$ 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 λ_{ZFC} -specific rules. Most simply use V_{set} to check tags before applying a lifted operator. The (image) rule replaces each value v_x in the set v_A with an application, generating a set expression, and the (set) rule reduces all the terms inside a set expression.

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 :

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

For example, $F_{\Downarrow}(\emptyset) = \{\langle v, v \rangle \mid v \in V\}$, by the (val) rule. $F_{\Downarrow}(F_{\Downarrow}(\emptyset))$ 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 γ . If λ_{ZFC} had only finite terms, $\gamma = \omega$ iterations would reach a fixpoint. But a simple countable term shows why “ \Downarrow ” cannot be F_{\Downarrow}^{ω} .

Example 3.10 (countably infinite term). If s is the successor function in λ_{ZFC} , the term $t := \langle t_{\text{set}}, \{0, \langle t_{\text{app}}, s, 0 \rangle, \langle t_{\text{app}}, s, \langle t_{\text{app}}, s, 0 \rangle, \dots \rangle\} \rangle$ should reduce to $\text{set}(\omega)$. The (set) rule’s premises require each of t ’s subterms to reduce—using at least F_{\Downarrow}^{ω} because each subterm requires a finite, unbounded number of (ap) derivations. Though $F_{\Downarrow}^{s(\omega)}$ reduces t , for larger terms, we must iterate F_{\Downarrow} much further. ◇

Theorem 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. □

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

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

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

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

3.6 Syntactic Sugar and a Small Set Library

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

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

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

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

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

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 λ_{ZFC} definitions. However, some deserve special mention.

Because λ_{ZFC} has only *functional* replacement, we cannot define unbounded “ \forall ” and

“ \exists ”. But we can define bounded quantifiers in terms of bounded selection, or

$$\begin{aligned} \text{select } f \ A &:= \bigcup (\text{image } (\lambda x. \text{if } (f x) \ {x} \not\in A) A) \\ \forall x \in e_A. e_f &:= (\text{select } (\lambda x. e_f) e_A) = e_A \\ \exists x \in e_A. e_f &:= (\text{select } (\lambda x. e_f) e_A) = \emptyset \end{aligned} \tag{3.12}$$

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

$$\iota x \in e_A. e_f := \text{take}(\text{select } (\lambda x. e_f) e_A) \tag{3.13}$$

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

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

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

defines ordered pairs, then

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

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

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

$$\begin{aligned} \text{return}_{\text{set}} \ a &:= \{a\} \\ \text{bind}_{\text{set}} \ A \ f &:= \bigcup (\text{image } f A) \end{aligned} \tag{3.16}$$

Using $\text{bind } m \ f = \text{join } (\text{lift } f m)$, evidently $\text{lift}_{\text{set}} := \text{image}$ and $\text{join}_{\text{set}} := \bigcup$. The proofs of the monad laws follow the proofs for the list monad. We also define

$$\{x \in e_A\}. e_f := \text{bind}_{\text{set}} (\lambda x. e_f) e_A \tag{3.17}$$

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

$$A \times B := \{x \in A\}. \{y \in B\}. \text{return}_{\text{set}} \langle x, y \rangle \quad (3.18)$$

Every $f \in A \rightarrow B$ is shaped $f = \{\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \dots\}$ and is total on A . To distinguish these hash tables from lambdas, we call them **mappings**. Mappings can be applied by $\text{ap } f x := \iota y \in (\text{range } f). \langle x, y \rangle \in f$, but we write just $f x$. We define

$$\text{mapping } f A := \text{image} (\lambda x. \langle x, f x \rangle) A \quad (3.19)$$

to convert a lambda to a mapping or to restrict a mapping to A . We usually use

$$\lambda x \in e_A. e_y := \text{mapping} (\lambda x. e_y) e_A \quad (3.20)$$

to define mappings.

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

3.7 Example: The Reals From the Rationals

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

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

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

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

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

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

So a sequence $\text{xs} \in \omega \rightarrow \mathbb{Q}$ converges to zero if, for any positive ε , there is some index N after which all xs are smaller than ε . Two sequences are equivalent ($=_R$) 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:

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

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

$$\begin{aligned}\text{quotient A } (=_A) &:= \text{image } (\lambda x. \text{select } (=_A x) A) A \\ \mathbb{R} &:= \text{quotient } R \ (=_R)\end{aligned}\tag{3.23}$$

defines the reals with extensional equality.

To define real arithmetic, we must lift rational arithmetic to sequences and then to sets of sequences. The map2 function lifts, say, $+_{\mathbb{Q}}$ to sequences, as in $(+)_R := \text{map2 } (+_{\mathbb{Q}})$. To lift $+_R$ to sets of sequences, note that sets of sequences are models of nondeterministic sequences, suggesting the set monad. We define $\text{lift2}_{\text{set}} f A B := \{a \in A\}. \{b \in B\}. \text{return}_{\text{set}} (f a b)$ to lift two-argument functions to the set monad. Now $(+) := \text{lift2}_{\text{set}} (+_R)$, and similarly for the other operators.

Using $\text{lift2}_{\text{set}}$ is atypical, so we prove that $A + B \in \mathbb{R}$ when $A \in \mathbb{R}$ and $B \in \mathbb{R}$, 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 $=_X$ is an equivalence relation on X , and define its quotient $\mathbb{X} := \text{quotient } X (=_X)$. If op is surjective on X and a binary morphism for $=_X$, then $(\text{lift2}_{\text{set}} \text{ op } A B) \in \mathbb{X}$ for all $A \in \mathbb{X}$ and $B \in \mathbb{X}$.*

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 $\text{conv-zero?}_{\mathbb{R}}$, which is like (3.21) but operates on real sequences $xs \in \omega \rightarrow \mathbb{R}$. We then define $\text{limit}_{\mathbb{R}} xs := \iota y \in \mathbb{R}. \text{conv-zero?}_{\mathbb{R}} (\text{map} (-y) xs)$ 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 λ_{ZFC} for expressing calculations involving limits, with bind_{lim} defined in terms of a general limit. We then derive a limit-free, computable replacement $\text{bind}'_{\text{lim}}$. Replacing bind_{lim} with $\text{bind}'_{\text{lim}}$ in a λ_{ZFC} term incurs proof obligations. If they can be met, the computable λ_{ZFC} term has the same limit as the original, uncomputable term.

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

The proof obligations are related to topological theorems [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 λ_{ZFC} . $A \Rightarrow B$ is a lambda or mapping type. $A \rightarrow B$ is the set of total mappings from A to 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 $A \subset \mathbb{U}$ then so is $\omega \rightarrow A$. Define \mathbb{U} as the language of $u ::= \mathbb{R} \mid \omega \rightarrow u$. A complete product metric $\delta : \mathbb{U} \Rightarrow \mathbb{U} \Rightarrow \mathbb{R}$ exists; therefore, a function $\text{limit} : (\omega \rightarrow \mathbb{U}) \Rightarrow \mathbb{U}$ similar to $\text{limit}_{\mathbb{R}}$ exists that calculates limits. These are all λ_{ZFC} -definable.

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

Example 3.14 (infinite series). Define $\text{partial-sums} : (\omega \rightarrow \mathbb{R}) \Rightarrow (\omega \rightarrow \mathbb{R})$ first by $\text{partial-sums}' xs := \lambda n. \text{if } (n = 0) (xs 0) ((xs n) + (\text{partial-sums}' xs (n - 1)))$. (The sequence is recursively defined, so we cannot use $\lambda n \in \omega. e$ to immediately create it.) Then convert it to a mapping: $\text{partial-sums} xs := \text{mapping}(\text{partial-sums}' xs) \omega$.

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

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

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

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

Example 3.15 (lifting). Define $\text{lift}_{\text{lim}} f xs := \text{bind}_{\text{lim}} xs \lambda x. \text{return}_{\text{lim}}(f x)$, as is typical. Substituting bind_{lim} and reducing reveals that $f(\text{limit} xs) = \text{limit}(\text{lift}_{\text{lim}} f xs)$. That is, using lift_{lim} pulls `limit` out of `f`'s argument. \diamond

Example 3.16 (exponential). The Taylor series expansion of the exponential function is $\text{exp-seq} : \mathbb{R} \Rightarrow (\omega \rightarrow \mathbb{R})$, defined by $\text{exp-seq} x := \text{partial-sums} \lambda n \in \omega. \frac{x^n}{n!}$. It always converges,

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

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 $\omega \rightarrow \mathbb{U}$, whose inhabitants are not directly implementable, so in the second step, we give them a lambda type.

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

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

Lemma 3.17 (continuity in metric spaces). *Let $f : A \Rightarrow B$ with A a metric space. Then f is continuous at $x \in A$ if and only if for all $\text{xs} \in \omega \rightarrow A$ for which $\text{limit} \text{xs} = x$ and f is defined on all elements of xs , $f(\text{limit} \text{xs}) = \text{limit}(f \text{xs})$.*

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

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

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

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

Lemma 3.18 (limits of sequences). *Let $f \in \omega \rightarrow \omega \rightarrow A$, where $\omega \rightarrow A$ has a product topology. Then $\text{limit } f = \lambda n \in \omega. \text{limit} (\text{flip } f n)$, where $\text{flip } f x y := f y x$.*

A countable product metric defines a product topology, so $\text{join}_{\text{lim}} xss := \lambda n \in \omega. \text{limit} (\text{flip } xss n)$.

Now we can remove `limit` by restricting `joinlim`'s input.

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

Lemma 3.20 (collapsing limits). *If $f \in \omega \rightarrow \omega \rightarrow \mathbb{U}$ converges uniformly, and $r, s : \omega \Rightarrow \omega$ increase, then $\text{limit } \lambda n \in \omega. \text{limit} (f n) = \text{limit } \lambda n \in \omega. f (r n) (s n)$.*

So if `flip xss` converges uniformly, then

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

We define $\text{join}'_{\text{lim}} : (\omega \rightarrow \omega \rightarrow \mathbb{U}) \Rightarrow (\omega \rightarrow \mathbb{U})$ by $\text{join}'_{\text{lim}} xss := \lambda n \in \omega. xss n n$. Replacing `joinlim xss` with `join'lim xss` requires that `flip xss` converge uniformly.

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

Example 3.21 (exponential cont.). Define exp'_{lim} by replacing `bindlim` by `bind'lim` in `explim`, so $\text{exp}'_{\text{lim}} xs := \text{bind}'_{\text{lim}} xs \text{exp-seq}$. We now meet the proof obligations.

Lemma 3.22. *Let $f : A \Rightarrow (\omega \rightarrow B)$. If $\omega \rightarrow B$ has a product topology, then f is continuous if and only if $(\text{flip } f) n$ is continuous for every $n \in \omega$.*

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

Example 3.23 (π). The definition of $\text{arctan}_{\text{lim}}$ is like exp_{lim} 's. Defining $\text{arctan}'_{\text{lim}}$, including proving correctness, is like defining exp'_{lim} . To compute π , we use

$$\begin{aligned}\pi_{\text{lim}} := & ((\text{return}_{\text{lim}} 16) \times_{\text{lim}} (\text{arctan}_{\text{lim}} (\text{return}_{\text{lim}} \frac{1}{5}))) -_{\text{lim}} \\ & ((\text{return}_{\text{lim}} 4) \times_{\text{lim}} (\text{arctan}_{\text{lim}} (\text{return}_{\text{lim}} \frac{1}{239})))\end{aligned}\quad (3.27)$$

where $(\cdot)_{\text{lim}}$ are lifted arithmetic operators. Because (3.27) does not directly use bind_{lim} , defining the limit-free π'_{lim} imposes no proof obligations. \diamond

In general, using functions defined in terms of $\text{bind}'_{\text{lim}}$ 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 $\omega \rightarrow \mathbb{U}$. After manipulating some terms, we define the final, computable limit monad by $\text{return}'_{\text{lim}} x := \lambda n. x$ and $\text{bind}'_{\text{lim}} xs f := \lambda n. f (xs n) n$. Computations have type $\omega \Rightarrow \mathbb{U}'$, where \mathbb{U}' contains countable sequences of rationals.

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

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 λ -calculus. Like λ_{ZFC} , interpreting them in set theory seems to require a slightly stronger theory than plain ZFC. HOL and CIC ensure consistency using types, and use the Curry-Howard correspondence to extract programs.

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

The key difference between λ_{ZFC} and these higher-order logics is that λ_{ZFC} is not a logic. It is a programming language with infinite terms, which by design includes a transitive model of set theory (Theorem 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 λ -calculus [25] has “infinitely deep” terms. Although it exists for investigating laziness, cyclic data, and undefinedness in finitary languages, it is possible to encode uncomputable mathematics in it. In λ_{ZFC} , such up-front encodings are unnecessary.

Hypercomputation [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 λ_{ZFC} might offer a civilized way to program such machines, we do not think of our work as hypercomputation, but as approaching computability from above.

3.10 Conclusions

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

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 λ_{ZFC}

The previous chapter defined λ_{ZFC} , an untyped, call-by-value, operational λ -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 λ_{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 λ_{ZFC} in a way that makes these characterizations absolutely precise.

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

4.1 Computations and Values

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

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

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

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

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

We could say that `count-from 0` does not terminate because it attempts “infinitely deep” computation. This limitation is necessary; for example, it prevents λ_{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:

$$\text{image } (\lambda n. n + 1) \mathbb{N} \quad \{n + 1 \mid n \in \mathbb{N}\} \quad (4.3)$$

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 λ_{ZFC} , all algebraic data structures are encoded as sets; e.g. the pair $\langle x, y \rangle$ can be encoded as $\{\{x\}, \{x, y\}\}$. Only the *existence* of encodings into sets is important, as it means data structures inherit a defining characteristic of sets: strictness. More precisely, 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 λ_{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.
- $x \Rightarrow y$ denotes a partial function.

- $\langle x, y \rangle$ denotes a pair of values with types x and y .
- $\text{Set } x$ denotes a set with members of type x .

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

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

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

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

$$X \rightsquigarrow_{\perp} Y ::= X \Rightarrow Y \cup \{\perp\} \quad (4.4)$$

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

4.3 Using ZFC Values and Theorems

Almost everything definable in ZFC can be defined by a finite λ_{ZFC} program. The previous chapter, for example, defined the real numbers, arithmetic, and limits. The only ZFC values that cannot be defined 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

¹Constructivists would object to allowing the law of excluded middle, which is derivable from λ_{ZFC} ’s if, and almost everyone else would object to allowing choice functions.

Almost all known ZFC theorems apply to λ_{ZFC} 's set values without alteration.² Proofs about λ_{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 $A : \text{Set} \times$ and $B : \text{Set} \times$, $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.*

Or, $A = B$ if and only if they contain the same members.

4.4 Internal Equality and External Equivalence

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

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

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 $\lambda a. 2$ is equivalent to $\lambda a. 1 + 1$. 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.

²The only exceptions are theorems that rely critically on the existence of an inaccessible cardinal.

³Assuming the existence of an inaccessible cardinal, which is a modest assumption, as $\text{ZFC}+\kappa$ 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 `image fst` means $\lambda A. \text{image } \text{fst } A$), binding forms such as indexed unions $\bigcup_{x \in e_A} e$, destructuring binds as in `swap` $\langle a, b \rangle := \langle b, a \rangle$, and comprehensions like $\{a \in A \mid a \in B\}$. 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:

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

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

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

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 λ_{ZFC} , lambda equality is decided by comparing body expressions, so lambdas are intensional.

In ZFC and λ_{ZFC} , function extensionality is achieved by encoding functions as sets of input-output pairs. For example, the increment function for the natural numbers is $\{\langle 0, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 3 \rangle, \dots\}$. (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. $(f\ a)$ or $f\ a$) to apply either kind.

Syntax for constructing unnamed mappings is defined by

$$\lambda x_a \in e_A. e_b := \text{mapping} (\lambda x_a. e_b) e_A \quad (4.6)$$

$$\begin{aligned} \text{mapping} : (X \Rightarrow Y) &\Rightarrow \text{Set } X \Rightarrow (X \multimap Y) \\ \text{mapping } f A &:= \text{image} (\lambda a. \langle a, f a \rangle) A \end{aligned} \quad (4.7)$$

For symmetry with partial functions $x \Rightarrow y$, `mapping` returns a member of the set $X \multimap Y$ of all *partial* mappings from X to Y ; i.e. if $g : X \multimap Y$, then g 's domain may be a subset of X . Two common partial mapping operations we use in the next chapter are

$$\begin{aligned} \text{domain} : (X \multimap Y) &\Rightarrow \text{Set } X \\ \text{domain } g &:= \text{image} \text{fst } g \end{aligned} \quad (4.8)$$

$$\begin{aligned} \text{preimage} : (X \multimap Y) &\Rightarrow \text{Set } Y \Rightarrow \text{Set } X \\ \text{preimage } g B &:= \{a \in \text{domain } g \mid g a \in B\} \end{aligned} \quad (4.9)$$

The `preimage` function finds g 's inputs whose corresponding outputs are in B .

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

In short, in addition to lambdas in λ_{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.

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 λ_{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 $\mathbf{P}[\cdot]$ and $\mathbf{D}[\cdot]$.

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** X as a total mapping

$$X : \Omega \rightarrow S_X \quad (5.1)$$

where Ω and S_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 S_X **observable outcomes**.

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

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

A tractable way defines $\Omega := \{1, 2, 3, 4, 5, 6\}$ and $E : \Omega \rightarrow \{\text{even}, \text{odd}\}$ so that $E \omega = \text{even}$ if $\omega \in \{2, 4, 6\}$, otherwise odd . The probability that $E \omega = \text{even}$ 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 $\Omega := \{\text{even}, \text{odd}\}$, each with probability $\frac{1}{2}$, and $E \omega := \omega$. \diamond

Random variables enable a kind of probabilistic abstraction. The example does it twice. The first makes calculating the probability that $E \omega = \text{even}$ tractable. The second is an

optimization. In fact, redefining Ω , 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 non-determinism 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 $A, B, C : \Omega \rightarrow \mathbb{R}$ are random variables, $C := A + B$ means $C \omega := (A \omega) + (B \omega)$, and $B := 4 + A$ means $B \omega := 4 + (A \omega)$.

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

$$\lambda \omega. ((\text{Plus } \omega) (\text{Four } \omega) (A \omega)) \quad (5.2)$$

We abstract lifting and application with these combinators:

$$\begin{aligned} \text{pure}_{\text{rv}} c &:= \lambda \omega. c \\ \text{ap}_{\text{rv}}^* F X_1 \dots X_n &:= \lambda \omega. ((F \omega) (X_1 \omega) \dots (X_n \omega)) \end{aligned} \quad (5.3)$$

$$\begin{aligned}
\mathcal{R}[X] &:= X \\
\mathcal{R}[x] &:= \text{pure}_{\text{rv}} x \\
\mathcal{R}[v] &:= \text{pure}_{\text{rv}} v \\
\mathcal{R}[e_f e_1 \dots e_n] &:= \text{ap}_{\text{rv}}^* \mathcal{R}[e_f] \mathcal{R}[e_1] \dots \mathcal{R}[e_n] \\
\mathcal{R}[\lambda x_1 \dots x_n. e] &:= \lambda \omega. \lambda x_1 \dots x_n. (\mathcal{R}[e] \omega) \\
\text{pure}_{\text{rv}} c &:= \lambda \omega. c \\
\text{ap}_{\text{rv}}^* F X_1 \dots X_n &:= \lambda \omega. ((F \omega) (X_1 \omega) \dots (X_n \omega))
\end{aligned}$$

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

In terms of pure_{rv} and ap_{rv}^* , $4 + A$ means

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

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 $I a ::= \Omega \Rightarrow a$ for some Ω .

$\mathcal{R}[\cdot]$ (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 ω . 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 **mapping** function, which, given a domain, converts a lambda or mapping to a mapping, as in **mapping** $\mathcal{R}[4 + A] \Omega$.

```

(define-syntax (RV/kernel stx)
  (syntax-parse stx
    [(_ Xs:id 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:id 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 Ω —except when Ω is countable. We thus deal here with **discrete probability measures** $P : \text{Set } \Omega \rightarrow [0, 1]$, where Ω is countable. Any discrete probability measure is uniquely determined by its value on singleton sets, or by a **probability mass function** $p : \Omega \rightarrow [0, 1]$. It is easy to convert p to a probability measure:

$$\text{sum } p A := \sum_{\omega \in A} p \omega \quad (5.5)$$

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

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

5.3.2 Background Theory: Queries

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

$$P \{\omega \in \Omega \mid E \omega = \text{even}\} = P \{2, 4, 6\} = \text{sum } p \{2, 4, 6\} = \frac{1}{2} \quad (5.6)$$

This is a **probability query**.

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

$$p_E \text{ even} = \text{sum } p(\text{preimage } E \{\text{even}\}) = \text{sum } p\{2, 4, 6\} = \frac{1}{2} \quad (5.7)$$

More abstractly, we can calculate discrete distribution queries using

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

Now $p_E = \text{dist } E \langle \Omega, p \rangle$. Recall that the special syntax $\lambda x \in e_A. e$ creates an unnamed mapping with domain e_A , and **mapping** $X \Omega$ converts X , which may be a lambda, to a mapping with domain Ω , 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 P and sets become propositions. For example, a common way to write “the probability of an even die outcome” in practice is $\Pr[E = \text{even}]$.

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

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

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

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 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 Ω to recursively enumerable sets. The implementation encodes them as lazy lists. We trust users to not create “sets” with duplicates.

When A is infinite, $\text{sum } p \ A$ is an infinite series. With 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 Ω and normalize p , which makes the sum finite and the distributions proper.

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

5.3.5 Implementation in Racket

Figure 5.3 shows the implementations of `finitize` and `dist` in Racket. The free variable 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 `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 semidecidable

```

(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.

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 Ω . 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 Ω .

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 P is discrete, however, the conditional probability of set A given set B (i.e. asserting that $\omega \in B$), simplifies to

$$\Pr[A | B] = P(A \cap B) / P B \quad (5.10)$$

In theory and practice, $\Pr[\cdot | \cdot]$ 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 $L : \Omega \rightarrow \{\text{low}, \text{high}\}$ defined by $L \omega = \text{if } (\omega \leq 3) \text{ low high}$. The probability that $E = \text{even}$ given $L = \text{low}$ is

$$\Pr[E = \text{even} | L = \text{low}] = \frac{\Pr[E = \text{even} \wedge L = \text{low}]}{\Pr[L = \text{low}]} = \frac{\sum_{\omega \in \{2\}} P \omega}{\sum_{\omega \in \{1,2,3\}} P \omega} = \frac{\frac{1}{6}}{\frac{1}{2}} = \frac{1}{3} \quad (5.11)$$

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

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 E given 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 Ω is restricted to the subset where random variable Y returns y :

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

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

We approximate **cond** by applying **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

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

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 L 's distribution is $p_L := [\text{low}, \text{high} \mapsto \frac{1}{2}]$, then E 's distribution depends on L 's output.

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

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

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

We can regard $L \sim p_L$ as a constraint on models: $\text{dist } L \langle \Omega, p \rangle$ must be p_L for every model $\langle \Omega, p, L \rangle$. Similarly, $E \sim (p_{E|L} L)$ means E 's conditional distribution is $p_{E|L}$. We have been using the model $\Omega := \{1, 2, 3, 4, 5, 6\}$, $p := [1, 2, 3, 4, 5, 6 \mapsto \frac{1}{6}]$, and the obvious E and L .

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

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

It is not hard to verify that this is also a model:

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

The construction of Ω , L and E in (5.15) clearly generalizes, but 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:

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

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

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

$$\begin{aligned}
 \Omega &:= S_1 \times \dots \times S_n \\
 X_i \langle \omega_1, \dots, \omega_i, \dots, \omega_n \rangle &:= \omega_i \quad (1 \leq i \leq n) \\
 p &:= \text{mapping } \mathcal{R}[(K_1 X_1) \cdot \dots \cdot (K_n X_n)] \ \Omega
 \end{aligned} \tag{5.17}$$

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 λ_{ZFC} term e_i whose interpretation is a discrete **transition kernel** $\mathcal{R}[e_i] : \Omega \Rightarrow S_i \rightarrow [0, 1]$. In measure theory,

```

distps X ⟨Ω, p⟩ := let SX := image X Ω
                      pX := λx ∈ SX. sum p (preimage (mapping X Ω) {x})
                  in ⟨Ω, p, pX⟩

condps Y y ⟨Ω, p⟩ := let Ω' := preimage (mapping Y Ω) {y}
                           p' := λω ∈ Ω'. (p ω) / (sum p Ω')
                       in ⟨Ω', p', _⟩

extendps K ⟨Ω, p⟩ := let S' ω := domain (K ω)
                           Ω' := (ω ∈ Ω) × (S' ω)
                           X ω := ωj (where j is the length of any ω ∈ Ω)
                           p' := mapping R[[p ∙ (K X)]] Ω'
                       in ⟨Ω', p', X⟩

emptyps := ⟨{⟨⟩}, λω ∈ {⟨⟩}. 1⟩

runps m := let ⟨Ω, p, x⟩ := m emptyps
              in x

```

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

transition kernels are used to build **product spaces** such as $\langle \Omega, p \rangle$. Thus, $\mathcal{R}[\cdot]$ links Bayesian practice to measure theory and represents an increase in expressive power in specifying distributions, by turning properly typed λ_{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

$$\begin{aligned} \text{return}_{\text{ps}} \times \langle \Omega, p \rangle &:= \langle \Omega, p, x \rangle \\ \text{bind}_{\text{ps}} m f \langle \Omega, p \rangle &:= \text{let } \langle \Omega', p', x \rangle := m \langle \Omega, p \rangle \\ &\quad \text{in } f x \langle \Omega', p' \rangle \end{aligned} \tag{5.18}$$

Figure 5.4 shows the additional `distps`, `condps` and `extendps`. The first two simply reimplement `dist` and `cond`. But `extendps`, which interprets statements, needs more explanation.

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

But this makes query implementation inefficient: if the union has little overlap or is disjoint, p will assign 0 to most ω . In more general terms, we actually have a *dependent cartesian product* $(\omega \in \Omega_{i-1}) \times (S'_i \omega)$, a generalization of the cartesian product.³ To extend Ω , `extendps` calculates this product instead.

Dependent cartesian products are elegantly expressed using the set monad:

$$\begin{aligned} \text{return}_{\text{set}} a &:= \{a\} \\ \text{bind}_{\text{set}} A f &:= \bigcup (\text{image } f A) \end{aligned} \tag{5.19}$$

Then $(a \in A) \times (B a) = \text{bind}_{\text{set}} A \lambda a. \text{bind}_{\text{set}} (B a) \lambda b. \text{return}_{\text{set}} \langle a, b \rangle$.

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 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 $\langle \rangle$. $\mathbf{D}[e_X | e_Y]$

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

$$\begin{aligned}
\mathcal{M}[X_i := e_i; t_{i+1}] &:= \text{bind}_{\text{ps}}(\text{return}_{\text{ps}} \mathcal{R}[e_i]) \lambda X_i. \mathcal{M}[t_{i+1}] \\
\mathcal{M}[X_i \sim e_i; t_{i+1}] &:= \text{bind}_{\text{ps}}(\text{extend}_{\text{ps}} \mathcal{R}[e_i]) \lambda X_i. \mathcal{M}[t_{i+1}] \\
\mathcal{M}[e_a = e_b; t_{i+1}] &:= \text{bind}_{\text{ps}}(\text{cond}_{\text{ps}} \mathcal{R}[e_a] \mathcal{R}[e_b]) \lambda _. \mathcal{M}[t_{i+1}] \\
\mathcal{M}[\epsilon] &:= \text{return}_{\text{ps}} \langle X_1, \dots, X_n \rangle \\
\\
\mathbf{D}[e] \ m &:= \text{run}_{\text{ps}}(\text{bind}_{\text{ps}} \ m \ \lambda \langle X_1, \dots, X_n \rangle. \text{dist}_{\text{ps}} \mathcal{R}[e]) \\
\mathbf{D}[e_X | e_Y] \ m &:= \lambda y. \mathbf{D}[e_X] (\text{bind}_{\text{ps}} \ m \ \lambda \langle X_1, \dots, X_n \rangle. \mathcal{M}[e_Y = y]) \\
\mathbf{P}[e] \ m &:= \mathbf{D}[e] \ m \ \text{true} \\
\mathbf{P}[e_A | e_B] \ m &:= \mathbf{D}[e_A | e_B] \ m \ \text{true} \ \text{true}
\end{aligned}$$

Figure 5.5: The conditional theory and query semantic functions.

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 dist_{ps} and cond_{ps} apply `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)) (\lambda (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 ~ P1] [E ~ (Pe/l L)])` expands to

```
(define-syntax die-roll #'(MDL [] ([L ~ P1] [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 $\text{appx-z} \geq 321$, 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 $t_1; \dots; t_n$ 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*

$$\mathcal{M}[t_1; \dots; t_i; t_{i+1}; \dots; t_n] \langle \Omega_0, p_0 \rangle = \mathcal{M}[t_1; \dots; t_{i+1}; t_i; \dots; t_n] \langle \Omega_0, p_0 \rangle.$$

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

The following theorem says \equiv_D is like observational equivalence with query contexts:

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

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

Theorem 5.10. The following table summarizes commutativity of cond_{ps} , dist_{ps} and $\text{extend}_{\text{ps}}$ in the probability space monad:

cond_{ps}	=		
$\text{extend}_{\text{ps}}$	=	$\equiv_{\mathbf{D}}$	
dist_{ps}	$\not\equiv_{\mathbf{D}}$	=	=
	cond_{ps}	$\text{extend}_{\text{ps}}$	dist_{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 λ_{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 λ_{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 λ_{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 $\langle \mathbb{R}, \Sigma, P \rangle$, where Σ is a subset of $\mathcal{P} \mathbb{R}$ and $P : \Sigma \rightarrow [0, 1]$.

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

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

To interpret `geometric p` using the uncountable probability space monad, we must interpret both branches of the 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 P assigns probabilities to subsets of X and $g : X \rightarrow Y$, then the distribution over subsets of Y is

$$Pr[B] = P(\text{preimage } g B) \quad (7.1)$$

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

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

$$Pr[B' \mid B] = Pr[B' \cap B] / Pr[B] \quad (7.2)$$

If $\Pr[B] = 0$, conditional probabilities can be calculated as the limit of $\Pr[B' | B_n]$ for certain positive-probability $B_1 \supseteq B_2 \supseteq B_3 \supseteq \dots$ whose intersection is B [46]. For example, if $Y = \mathbb{R} \times \mathbb{R}$, the distribution of $\langle x, y \rangle \in Y$ given $x + y = 0$ can be calculated using the descending sequence $B_n = \{\langle x, y \rangle \in Y \mid |x + y| < 2^{-n}\}$.

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

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 $g : X \rightarrow Y$, the probability measure on output sets $B \subseteq Y$ should be defined by preimages of B under g and the probability measure on X . Unfortunately, it is difficult to turn this simple-sounding idea into a compositional semantics, for the following reasons.

1. Preimages are definable only for functions with observable domains, which excludes lambdas.
2. If subsets of X and Y must be measurable, taking preimages under g must preserve measurability (we say g itself is measurable). Proving the conditions under which this is true is difficult, especially if 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 λ_{ZFC} [49], a λ -calculus with infinite sets, and both extensional and intensional functions. We address 5 by deriving and implementing a *conservative approximation* of the semantics.

There seems to be no way to simplify difficulty 2, so we work through 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 Appendix A.

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:

$$\begin{array}{ccccc}
 X \rightsquigarrow_{\perp} Y & \xrightarrow{\text{lift}_{\text{map}}} & X \rightsquigarrow_{\text{map}} Y & \xrightarrow{\text{lift}_{\text{pre}}} & X \rightsquigarrow_{\text{pre}} Y \\
 \eta_{\perp^*} \downarrow & & \downarrow \eta_{\text{map}^*} & & \downarrow \eta_{\text{pre}^*} \\
 X \rightsquigarrow_{\perp^*} Y & \xrightarrow{\text{lift}_{\text{map}^*}} & X \rightsquigarrow_{\text{map}^*} Y & \xrightarrow{\text{lift}_{\text{pre}^*}} & X \rightsquigarrow_{\text{pre}^*} Y
 \end{array} \tag{7.3}$$

At the top-left, $X \rightsquigarrow_{\perp} Y$ 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 lift_{map} combinator produces $X \rightsquigarrow_{\text{map}} Y$ computations, which create equivalent extensional functions, or mappings. From mapping arrow computations, the lift_{pre} combinator produces $X \rightsquigarrow_{\text{pre}} Y$ 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 lift_{map} distributes over all bottom arrow combinators.

The approximating semantics uses the same semantic function, but its arrows $X \rightsquigarrow'_{\text{pre}} Y$ and $X \rightsquigarrow'^*_{\text{pre}} Y$ 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 [55] 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 $x \rightsquigarrow_a y$ must behave like a corresponding function of type $x \Rightarrow y$ (in a sense we explain shortly).
- First-order: There is no way to derive a computation $\text{app}_a : \langle x \rightsquigarrow_a y, x \rangle \rightsquigarrow_a y$ from the arrow 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 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 a , instead of first_a , we define $(\&\&_a)$. This combinator is typically called **fanout**, but its use will be clearer if we call it **pairing**. One way to strengthen an arrow a is to define an additional combinator left_a , which can be used to choose an arrow computation based on the result of another. Again, we define a different combinator, ifte_a (“if-then-else”).

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

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

and try to define the following recursive function:

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

In a strict λ -calculus, the defining expression does not terminate. But the following is

well-defined in λ_{ZFC} , and loops only when applied to `false`:

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

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

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

$$\begin{aligned}
 \text{arr}_a : (x \Rightarrow y) \Rightarrow (x \rightsquigarrow_a y) && \text{lift} \\
 (\ggg_a) : (x \rightsquigarrow_a y) \Rightarrow (y \rightsquigarrow_a z) \Rightarrow (x \rightsquigarrow_a z) && \text{composition} \\
 (\&&_a) : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_a z) \Rightarrow (x \rightsquigarrow_a \langle y, z \rangle) && \text{pairing} \\
 \text{ifte}_a : (x \rightsquigarrow_a \text{Bool}) \Rightarrow (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_a y) && \text{if-then-else} \\
 \text{lazy}_a : (1 \Rightarrow (x \rightsquigarrow_a y)) \Rightarrow (x \rightsquigarrow_a y) && \text{laziness}
 \end{aligned} \tag{7.7}$$

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 $\text{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$ is an arrow homomorphism from arrow a to arrow b if the following distributive laws hold for*

appropriately typed f , f_1 , f_2 and f_3 :

$$\text{lift}_b (\text{arr}_a f) \equiv \text{arr}_b f \quad (7.8)$$

$$\text{lift}_b (f_1 \ggg_a f_2) \equiv (\text{lift}_b f_1) \ggg_b (\text{lift}_b f_2) \quad (7.9)$$

$$\text{lift}_b (f_1 \&&_a f_2) \equiv (\text{lift}_b f_1) \&&_b (\text{lift}_b f_2) \quad (7.10)$$

$$\text{lift}_b (\text{ifte}_a f_1 f_2 f_3) \equiv \text{ifte}_b (\text{lift}_b f_1) (\text{lift}_b f_2) (\text{lift}_b f_3) \quad (7.11)$$

$$\text{lift}_b (\text{lazy}_a f) \equiv \text{lazy}_b \lambda 0. \text{lift}_b (f 0) \quad (7.12)$$

The arrow homomorphism laws state that $\text{arr}_a : (x \Rightarrow y) \Rightarrow (x \rightsquigarrow_a y)$ must be a homomorphism from the function arrow (7.4) to arrow a . Roughly, arrow computations that do not use additional combinators can be transformed into 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_a) and a pair extraction law:

$$(f_1 \ggg_a f_2) \ggg_a f_3 \equiv f_1 \ggg_a (f_2 \ggg_a f_3) \quad (7.13)$$

$$(\text{arr}_a f_1 \&&_a f_2) \ggg_a \text{arr}_a \text{snd} \equiv f_2 \quad (7.14)$$

and distribution of pure computations over effectful:

$$\text{arr}_a f_1 \ggg_a (f_2 \&&_a f_3) \equiv (\text{arr}_a f_1 \ggg_a f_2) \&&_a (\text{arr}_a f_1 \ggg_a f_3) \quad (7.15)$$

$$\begin{aligned} \text{arr}_a f_1 \ggg_a \text{ifte}_a f_2 f_3 f_4 &\equiv \text{ifte}_a (\text{arr}_a f_1 \ggg_a f_2) \\ &\quad (\text{arr}_a f_1 \ggg_a f_3) \\ &\quad (\text{arr}_a f_1 \ggg_a f_4) \end{aligned} \quad (7.16)$$

$$\text{arr}_a f_1 \ggg_a \text{lazy}_a f_2 \equiv \text{lazy}_a \lambda 0. \text{arr}_a f_1 \ggg_a f_2 0 \quad (7.17)$$

Equivalence between different arrow representations is usually proved in a strongly normalizing λ -calculus [31, 32], in which every function is free of effects, including nontermination. Such a λ -calculus has no need for 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 λ -calculi as well as λ_{ZFC} . If 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 f_1 in the pair extraction law to always terminate. Instead, we require every argument to arr_a to terminate, which simplifies more proofs.

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

Definition 7.3 (arrow epimorphism). *An arrow homomorphism $\text{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$ that has a right inverse is an **arrow epimorphism** from a to b .*

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

Proof. Let lift_b^{-1} be lift_b 's right inverse. For the pair extraction law (7.14),

$$\begin{aligned}
 & (\text{arr}_b f_1 \&\&_b f_2) \ggg_b \text{arr}_b \text{snd} && (7.18) \\
 & \equiv (\text{lift}_b (\text{arr}_a f_1) \&\&_b (\text{lift}_b (\text{lift}_b^{-1} f_2))) \ggg_b \text{lift}_b (\text{arr}_a \text{snd}) && \text{Rewrite with } \text{lift}_b \\
 & \equiv \text{lift}_b (\text{arr}_a f_1 \&\&_a \text{lift}_b^{-1} f_2) \ggg_b \text{lift}_b (\text{arr}_a \text{snd}) && \text{Homomorphism (7.10)} \\
 & \equiv \text{lift}_b ((\text{arr}_a f_1 \&\&_a \text{lift}_b^{-1} f_2) \ggg_a \text{arr}_a \text{snd}) && \text{Homomorphism (7.9)} \\
 & \equiv \text{lift}_b (\text{lift}_b^{-1} f_2) && \text{Pair extraction (7.14)} \\
 & \equiv f_2 && \text{Right inverse}
 \end{aligned}$$

The proofs for every other law are similar. \square

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.

$p ::= x := e; \dots ; e$	$e ::= x e \mid \text{let } e e \mid \text{env } n \mid \langle e, e \rangle \mid \text{fst } e \mid \text{snd } e \mid \text{if } e e e \mid v$
$v ::= \text{[first-order constants]}$	
$\llbracket x := e; \dots ; e_b \rrbracket_a ::= x := \llbracket e \rrbracket_a; \dots ; \llbracket e_b \rrbracket_a$	
$\llbracket x e \rrbracket_a ::= \llbracket \langle e, \rangle \rrbracket_a \ggg_a x$	$\llbracket \text{let } e e_b \rrbracket_a ::= (\llbracket e \rrbracket_a \&&_a \text{arr}_a \text{id}) \ggg_a \llbracket e_b \rrbracket_a$
$\llbracket \langle e_1, e_2 \rangle \rrbracket_a ::= \llbracket e_1 \rrbracket_a \&&_a \llbracket e_2 \rrbracket_a$	$\llbracket \text{env } 0 \rrbracket_a ::= \text{arr}_a \text{fst}$
$\llbracket \text{fst } e \rrbracket_a ::= \llbracket e \rrbracket_a \ggg_a \text{arr}_a \text{fst}$	$\llbracket \text{env } (n+1) \rrbracket_a ::= \text{arr}_a \text{snd} \ggg_a \llbracket \text{env } n \rrbracket_a$
$\llbracket \text{snd } e \rrbracket_a ::= \llbracket e \rrbracket_a \ggg_a \text{arr}_a \text{snd}$	$\llbracket \text{if } e_c e_t e_f \rrbracket_a ::= \text{ifte}_a \llbracket e_c \rrbracket_a \llbracket \text{lazy } e_t \rrbracket_a \llbracket \text{lazy } e_f \rrbracket_a$
$\llbracket v \rrbracket_a ::= \text{arr}_a (\text{const } v)$	$\llbracket \text{lazy } e \rrbracket_a ::= \text{lazy}_a \lambda 0. \llbracket e \rrbracket_a$
$\text{id} ::= \lambda a. a$	
$\text{const } b ::= \lambda a. b$	subject to $\llbracket p \rrbracket_a : \langle \rangle \rightsquigarrow_a y$ for some y

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

The semantic function $\llbracket \cdot \rrbracket_a$ 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 $\langle \rangle \rightsquigarrow_a y$, where y is the type of the program's value, and $\langle \rangle$ denotes an empty list, or stack. A `let` expression pushes a value onto the stack. First-order functions have type $\langle x, \rangle \rightsquigarrow_a y$ where x is the argument type and y is the return type. Application sends a stack containing just an x .

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

Definition 7.5 (well-defined expression). *An expression e is **well-defined** under arrow a if $\llbracket e \rrbracket_a : x \rightsquigarrow_a y$ for some x and y , and $\llbracket e \rrbracket_a$ terminates.*

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

our semantic correctness results rely.

Theorem 7.6 (homomorphisms distribute over expressions). *Let $\text{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$ be an arrow homomorphism. For all e , $\llbracket e \rrbracket_b \equiv \text{lift}_b \llbracket e \rrbracket_a$.*

Proof. By structural induction. Base cases proceed by expansion and using $\text{arr}_b \equiv \text{lift}_b \circ \text{arr}_a$ (7.8). For example, for constants:

$$\begin{aligned} \llbracket v \rrbracket_b &\equiv \text{arr}_b (\text{const } v) && \text{Def of } \llbracket \cdot \rrbracket_b \\ &\equiv \text{lift}_b (\text{arr}_a (\text{const } v)) && \text{Homomorphism (7.8)} \\ &\equiv \text{lift}_b \llbracket v \rrbracket_a && \text{Def of } \llbracket \cdot \rrbracket_a \end{aligned} \tag{7.19}$$

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

$$\begin{aligned} \llbracket (e_1, e_2) \rrbracket_b &\equiv \llbracket e_1 \rrbracket_b \&\&_b \llbracket e_2 \rrbracket_b && \text{Def of } \llbracket \cdot \rrbracket_b \\ &\equiv (\text{lift}_b \llbracket e_1 \rrbracket_a) \&\&_b (\text{lift}_b \llbracket e_2 \rrbracket_a) && \text{Ind hypothesis} \\ &\equiv \text{lift}_b (\llbracket e_1 \rrbracket_a \&\&_a \llbracket e_2 \rrbracket_a) && \text{Homomorphism (7.10)} \\ &\equiv \text{lift}_b \llbracket (e_1, e_2) \rrbracket_a && \text{Def of } \llbracket \cdot \rrbracket_a \end{aligned} \tag{7.20}$$

It is not hard to check the remaining cases. \square

If we assume lift_b defines correct behavior for arrow b in terms of arrow a , and prove that lift_b is a homomorphism, then by Theorem 7.6, $\llbracket \cdot \rrbracket_b$ is correct.

7.3 The Bottom Arrow

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

$$\begin{array}{ccccc} X \rightsquigarrow_{\perp} Y & \xrightarrow{\text{lift}_{\text{map}}} & X \rightsquigarrow_{\text{map}} Y & \xrightarrow{\text{lift}_{\text{pre}}} & X \rightsquigarrow_{\text{pre}} Y \\ \eta_{\perp^*} \downarrow & & \downarrow \eta_{\text{map}^*} & & \downarrow \eta_{\text{pre}^*} \\ X \rightsquigarrow_{\perp^*} Y & \xrightarrow{\text{lift}_{\text{map}^*}} & X \rightsquigarrow_{\text{map}^*} Y & \xrightarrow{\text{lift}_{\text{pre}^*}} & X \rightsquigarrow_{\text{pre}^*} Y \end{array} \tag{7.21}$$

$X \rightsquigarrow_{\perp} Y ::= X \Rightarrow Y_{\perp}$ $\text{arr}_{\perp} : (X \Rightarrow Y) \Rightarrow (X \rightsquigarrow_{\perp} Y)$ $\text{arr}_{\perp} f := f$ $(\ggg_{\perp}) : (X \rightsquigarrow_{\perp} Y) \Rightarrow (Y \rightsquigarrow_{\perp} Z) \Rightarrow (X \rightsquigarrow_{\perp} Z)$ $(f_1 \ggg_{\perp} f_2) a := \text{if } (f_1 a = \perp) \perp (f_2 (f_1 a))$ $(\&&_{\perp}) : (X \rightsquigarrow_{\perp} Y_1) \Rightarrow (X \rightsquigarrow_{\perp} Y_2) \Rightarrow (X \rightsquigarrow_{\perp} \langle Y_1, Y_2 \rangle)$ $(f_1 \&&_{\perp} f_2) a := \text{let } b_1 := f_1 a$ $\quad \quad \quad b_2 := f_2 a$ $\quad \quad \quad \text{in if } (b_1 = \perp \text{ or } b_2 = \perp) \perp \langle b_1, b_2 \rangle$	$\text{ifte}_{\perp} : (X \rightsquigarrow_{\perp} \text{Bool}) \Rightarrow (X \rightsquigarrow_{\perp} Y) \Rightarrow (X \rightsquigarrow_{\perp} Y) \Rightarrow (X \rightsquigarrow_{\perp} Y)$ $\text{ifte}_{\perp} f_1 f_2 f_3 a := \text{case } f_1 a$ $\quad \quad \quad \text{true} \longrightarrow f_2 a$ $\quad \quad \quad \text{false} \longrightarrow f_3 a$ $\quad \quad \quad \perp \longrightarrow \perp$ $\text{lazy}_{\perp} : (1 \Rightarrow (X \rightsquigarrow_{\perp} Y)) \Rightarrow (X \rightsquigarrow_{\perp} Y)$ $\text{lazy}_{\perp} f a := f 0 a$
---	---

Figure 7.2: Bottom arrow definitions.

Through Section 7.6, we move across the top to $X \xrightarrow{\text{pre}} Y$.

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 $X \rightsquigarrow_{\perp} Y ::= X \Rightarrow Y_{\perp}$, where $Y_{\perp} ::= Y \cup \{\perp\}$ and \perp is a distinguished error value. The type Bool_{\perp} , for example, denotes the members of $\text{Bool} \cup \{\perp\} = \{\text{true}, \text{false}, \perp\}$.

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

Definition 7.7 (bottom arrow equivalence). *Two computations $f_1 : X \rightsquigarrow_{\perp} Y$ and $f_2 : X \rightsquigarrow_{\perp} Y$ are equivalent, or $f_1 \equiv f_2$, when $f_1 a \equiv f_2 a$ for all $a \in X$.*

Theorem 7.8. arr_{\perp} , $(\&&_{\perp})$, (\ggg_{\perp}) , ifte_{\perp} and lazy_{\perp} define an arrow.

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

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 an 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. $X \rightsquigarrow_{\text{map}} Y ::= X \multimap Y$. Unfortunately, we could not define $\text{arr}_{\text{map}} : (X \Rightarrow Y) \Rightarrow (X \multimap Y)$: 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

$$X \rightsquigarrow_{\text{map}} Y ::= \text{Set } X \Rightarrow (X \multimap Y) \quad (7.22)$$

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

To use Theorem 7.6 to prove that expressions interpreted using $\llbracket \cdot \rrbracket_{\text{map}}$ behave correctly with respect to $\llbracket \cdot \rrbracket_{\perp}$, we need to define correctness using a lift from the bottom arrow to the mapping arrow. It is helpful to have a standalone function domain_{\perp} that computes the subset of A on which f does not return \perp . We define that first, and then define lift_{map} in terms of it:

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

$$\begin{aligned} \text{lift}_{\text{map}} &: (X \rightsquigarrow_{\perp} Y) \Rightarrow (X \rightsquigarrow_{\text{map}} Y) \\ \text{lift}_{\text{map}} f A &:= \text{mapping } f (\text{domain}_{\perp} f A) \end{aligned} \quad (7.24)$$

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

$\text{range} : (X \multimap Y) \Rightarrow \text{Set } Y$	$\langle \cdot, \cdot \rangle_{\text{map}} : (X \multimap Y_1) \Rightarrow (X \multimap Y_2) \Rightarrow (X \multimap Y_1 \times Y_2)$
$\text{range } g := \text{image } \text{snd } g$	$\langle g_1, g_2 \rangle_{\text{map}} := \text{let } A := \text{domain } g_1 \cap \text{domain } g_2$
$(\circ_{\text{map}}) : (Y \multimap Z) \Rightarrow (X \multimap Y) \Rightarrow (X \multimap Z)$	$\text{in } \lambda a \in A. (g_1 a, g_2 a)$
$g_2 \circ_{\text{map}} g_1 := \text{let } A := \text{preimage } g_1 (\text{domain } g_2)$	$(\sqcup_{\text{map}}) : (X \multimap Y) \Rightarrow (X \multimap Y) \Rightarrow (X \multimap Y)$
$\text{in } \lambda a \in A. g_2 (g_1 a)$	$g_1 \sqcup_{\text{map}} g_2 := \text{let } A := \text{domain } g_1 \sqcup \text{domain } g_2$
	$\text{in } \lambda a \in A. \text{if } (a \in \text{domain } g_1) (g_1 a) (g_2 a)$

Figure 7.3: Additional operations on partial mappings.

$X_{\text{map}} \rightsquigarrow Y ::= \text{Set } X \Rightarrow (X \multimap Y)$	$\text{ifte}_{\text{map}} : (X_{\text{map}} \text{ Bool}) \Rightarrow (X_{\text{map}} Y) \Rightarrow (X_{\text{map}} Y) \Rightarrow (X_{\text{map}} Y)$
$\text{arr}_{\text{map}} : (X \Rightarrow Y) \Rightarrow (X_{\text{map}} \rightsquigarrow Y)$	$\text{ifte}_{\text{map}} g_1 g_2 g_3 A := \text{let } g'_1 := g_1 A$
$\text{arr}_{\text{map}} := \text{lift}_{\text{map}} \circ \text{arr}_{\perp}$	$g'_2 := g_2 (\text{preimage } g'_1 \{ \text{true} \})$
$(\ggg_{\text{map}}) : (X_{\text{map}} Y) \Rightarrow (Y_{\text{map}} Z) \Rightarrow (X_{\text{map}} Z)$	$g'_3 := g_3 (\text{preimage } g'_1 \{ \text{false} \})$
$(g_1 \ggg_{\text{map}} g_2) A := \text{let } g'_1 := g_1 A$	$\text{in } g'_2 \sqcup_{\text{map}} g'_3$
$g'_2 := g_2 (\text{range } g'_1)$	
$\text{in } g'_2 \circ_{\text{map}} g'_1$	
$(\&&_{\text{map}}) : (X_{\text{map}} Y_1) \Rightarrow (X_{\text{map}} Y_2) \Rightarrow (X_{\text{map}} \langle Y_1, Y_2 \rangle)$	$\text{lazy}_{\text{map}} : (1 \Rightarrow (X_{\text{map}} Y)) \Rightarrow (X_{\text{map}} Y)$
$(g_1 \&&_{\text{map}} g_2) A := \langle g_1 A, g_2 A \rangle_{\text{map}}$	$\text{lazy}_{\text{map}} g A := \text{if } (A = \emptyset) \emptyset (g 0 A)$
	<hr/>
	$\text{lift}_{\text{map}} : (X \rightsquigarrow_{\perp} Y) \Rightarrow (X_{\text{map}} \rightsquigarrow Y)$
	$\text{lift}_{\text{map}} f A := \{ \langle a, b \rangle \in \text{mapping } f A \mid b \neq \perp \}$

Figure 7.4: Mapping arrow definitions.

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

Definition 7.9 (mapping arrow equivalence). *Two computations $g_1 : X_{\text{map}} \rightsquigarrow Y$ and $g_2 : X_{\text{map}} \rightsquigarrow Y$ are equivalent, or $g_1 \equiv g_2$, when $g_1 A \equiv g_2 A$ for all $A \subseteq X$.*

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

7.4.1 Composition

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

$$\begin{aligned}
& \text{lift}_{\text{map}} (f_1 \ggg_{\perp} f_2) A && (7.25) \\
&\equiv \text{let } f := \lambda a. \text{if } (f_1 a = \perp) \perp (f_2 (f_1 a)) && \text{Def of lift}_{\text{map}}, (\ggg_{\perp}) \\
&\quad A' := \text{domain}_{\perp} f A \\
&\quad \text{in } \text{mapping } f A' \\
&\equiv \text{let } f := \lambda a. f_2 (f_1 a) && \text{Simplify } f \\
&\quad A' := \text{domain}_{\perp} f (\text{domain}_{\perp} f_1 A) \\
&\quad \text{in } \text{mapping } f A' \\
&\equiv \text{let } A' := \{a \in \text{domain}_{\perp} f_1 A \mid f_2 (f_1 a) \neq \perp\} && \text{Def of domain}_{\perp}, \text{ mapping} \\
&\quad \text{in } \lambda a \in A'. f_2 (f_1 a)
\end{aligned}$$

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

$$\begin{aligned}
&\equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A && \text{Rewrite with lift}_{\text{map}} \\
&\quad A' := \text{preimage } g_1 (\text{domain}_{\perp} f_2 (\text{range } g_1)) \\
&\quad \text{in } \lambda a \in A'. f_2 (g_1 a) \\
&\equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A && \text{Rewrite with lift}_{\text{map}} \\
&\quad g_2 := \text{lift}_{\text{map}} f_2 (\text{range } g_1) \\
&\quad A' := \text{preimage } g_1 (\text{domain } g_2) \\
&\quad \text{in } \lambda a \in A'. g_2 (g_1 a) \\
&\equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A && \text{Rewrite with } (\circ_{\text{map}}) \\
&\quad g_2 := \text{lift}_{\text{map}} f_2 (\text{range } g_1) \\
&\quad \text{in } g_2 \circ_{\text{map}} g_1
\end{aligned}$$

Substituting g_1 for $\text{lift}_{\text{map}} f_1$ and g_2 for $\text{lift}_{\text{map}} f_2$ gives a definition for (\ggg_{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 f by restricting it to a set for which $f_1 a \neq \perp$ and $f_2 a \neq \perp$:

$$\begin{aligned}
& \text{lift}_{\text{map}} (f_1 \&\&_{\perp} f_2) A && (7.26) \\
\equiv & \text{let } f := \lambda a. \text{let } b_1 := f_1 a && \text{Def of lift}_{\text{map}}, (\&\&_{\perp}) \\
& \quad b_2 := f_2 a \\
& \quad \text{in if } (b_1 = \perp \text{ or } b_2 = \perp) \perp \langle b_1, b_2 \rangle \\
& \quad A' := \text{domain}_{\perp} f A \\
& \quad \text{in mapping } f A' \\
\equiv & \text{let } f := \lambda a. \langle f_1 a, f_2 a \rangle && \text{Simplify } f \\
& A' := \text{domain}_{\perp} f_1 A \cap \text{domain}_{\perp} f_2 A \\
& \text{in mapping } f A' \\
\equiv & \text{let } A' := \text{domain}_{\perp} f_1 A \cap \text{domain}_{\perp} f_2 A && \text{Def of mapping} \\
& \text{in } \lambda a \in A'. \langle f_1 a, f_2 a \rangle
\end{aligned}$$

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

$$\begin{aligned}
& \equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A && \text{Rewrite with lift}_{\text{map}} \\
& \quad g_2 := \text{lift}_{\text{map}} f_2 A \\
& \quad A' := \text{domain } g_1 \cap \text{domain } g_2 \\
& \quad \text{in } \lambda a \in A'. \langle g_1 a, g_2 a \rangle \\
& \equiv \langle \text{lift}_{\text{map}} f_1 A, \text{lift}_{\text{map}} f_2 A \rangle_{\text{map}} && \text{Rewrite with } \langle \cdot, \cdot \rangle_{\text{map}}
\end{aligned}$$

Substituting g_1 for $\text{lift}_{\text{map}} f_1$ and g_2 for $\text{lift}_{\text{map}} f_2$ gives a definition for $(\&\&_{\text{map}})$ (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 f by restricting it to a domain for which $f_1 a \neq \perp$:

$$\text{lift}_{\text{map}} (\text{ifte}_{\perp} f_1 f_2 f_3) A \quad (7.27)$$

$$\begin{aligned}
 &\equiv \text{let } f := \lambda a. \text{case } f_1 a \\
 &\quad \text{true} \longrightarrow f_2 a \\
 &\quad \text{false} \longrightarrow f_3 a \\
 &\quad \perp \longrightarrow \perp \\
 &A' := \text{domain}_{\perp} f A \\
 &\text{in mapping } f A' \quad \text{Def of lift}_{\text{map}}, \text{ ifte}_{\perp} \\
 \\
 &\equiv \text{let } f := \lambda a. \text{if } (f_1 a) (f_2 a) (f_3 a) \quad \text{Simplify } f \\
 &\quad g_1 := \text{mapping } f_1 (\text{domain}_{\perp} f_1 A) \\
 &\quad A_2 := \text{preimage } g_1 \{\text{true}\} \\
 &\quad A_3 := \text{preimage } g_1 \{\text{false}\} \\
 &\quad A' := \text{domain}_{\perp} f_2 A_2 \uplus \text{domain}_{\perp} f_3 A_3 \\
 &\text{in mapping } f A' \\
 \\
 &\equiv \text{let } g_1 := \text{mapping } f_1 (\text{domain}_{\perp} f_1 A) \quad \text{Def of mapping} \\
 &\quad A_2 := \text{preimage } g_1 \{\text{true}\} \\
 &\quad A_3 := \text{preimage } g_1 \{\text{false}\} \\
 &\quad A' := \text{domain}_{\perp} f_2 A_2 \uplus \text{domain}_{\perp} f_3 A_3 \\
 &\text{in } \lambda a \in A'. \text{if } (f_1 a) (f_2 a) (f_3 a)
 \end{aligned}$$

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

$$\begin{aligned}
 &\equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A \quad \text{Rewrite with lift}_{\text{map}} \\
 &\quad g_2 := \text{lift}_{\text{map}} f_2 (\text{preimage } g_1 \{\text{true}\}) \\
 &\quad g_3 := \text{lift}_{\text{map}} f_3 (\text{preimage } g_1 \{\text{false}\}) \\
 &\quad A' := \text{domain } g_2 \uplus \text{domain } g_3 \\
 &\text{in } \lambda a \in A'. \text{if } (a \in \text{domain } g_2) (g_2 a) (g_3 a) \\
 \\
 &\equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A \quad \text{Rewrite with } (\uplus_{\text{map}}) \\
 &\quad g_2 := \text{lift}_{\text{map}} f_2 (\text{preimage } g_1 \{\text{true}\}) \\
 &\quad g_3 := \text{lift}_{\text{map}} f_3 (\text{preimage } g_1 \{\text{false}\}) \\
 &\text{in } g_2 \uplus_{\text{map}} g_3
 \end{aligned}$$

Substituting g_1 for $\text{lift}_{\text{map}} f_1$, g_2 for $\text{lift}_{\text{map}} f_2$, and g_3 for $\text{lift}_{\text{map}} f_3$ gives a definition for ifte_{map} (Figure 7.4) for which (7.11) holds.

7.4.4 Laziness

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

$$\text{lift}_{\text{map}} (\text{lazy}_\perp f) A \equiv \text{let } A' := \text{domain}_\perp (\lambda a. f 0 a) A \text{ in } \text{mapping} (\lambda a. f 0 a) A' \quad (7.28)$$

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

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

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

7.4.5 Correctness

Theorem 7.10 (mapping arrow correctness). lift_{map} is a homomorphism.

Proof. By construction. □

Corollary 7.11 (semantic correctness). For all e , $\llbracket e \rrbracket_{\text{map}} \equiv \text{lift}_{\text{map}} \llbracket e \rrbracket_\perp$.

Without restrictions, mapping arrow computations can be quite unruly. For example, the following computation is well-typed, but returns the identity mapping on `Bool` when

applied to an empty domain, and the empty mapping when applied to any other domain:

$$\begin{aligned} \text{nonmonotone} : \text{Bool} &\rightsquigarrow_{\text{map}} \text{Bool} \\ \text{nonmonotone } A := & \text{ if } (A = \emptyset) (\lambda a \in \text{Bool}. a) \emptyset \end{aligned} \tag{7.30}$$

It would be nice if we could be sure that every $X \rightsquigarrow_{\text{map}} Y$ 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 $g : X \rightsquigarrow_{\text{map}} Y$. If there exists an $f : X \rightsquigarrow_{\perp} Y$ such that $g \equiv \text{lift}_{\text{map}} f$, then g obeys the **mapping arrow law**.*

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

Theorem 7.13. lift_{map} is an arrow epimorphism.

Proof. Follows from Theorem 7.10 and restriction of $X \rightsquigarrow_{\text{map}} Y$ to instances for which the mapping arrow law (Definition 7.12) holds. \square

Corollary 7.14. arr_{map} , $(\&&_{\text{map}})$, (\ggg_{map}) , ifte_{map} and lazy_{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.

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

$$X \xrightarrow{\text{pre}} Y ::= \langle \text{Set } Y, \text{Set } Y \Rightarrow \text{Set } X \rangle \tag{7.31}$$

$X \xrightarrow{\text{pre}} Y ::= \langle \text{Set } Y, \text{Set } Y \Rightarrow \text{Set } X \rangle$	$\langle \cdot, \cdot \rangle_{\text{pre}} : (X \xrightarrow{\text{pre}} Y_1) \Rightarrow (X \xrightarrow{\text{pre}} Y_2) \Rightarrow (X \xrightarrow{\text{pre}} Y_1 \times Y_2)$
$\text{pre} : (X \xrightarrow{\text{map}} Y) \Rightarrow (X \xrightarrow{\text{pre}} Y)$	$\langle \langle Y'_1, p_1 \rangle, \langle Y'_2, p_2 \rangle \rangle_{\text{pre}} := \text{let } Y' := Y'_1 \times Y'_2$
$\text{pre } g := \langle \text{range } g, \lambda B. \text{preimage } g B \rangle$	$p := \lambda B. \bigcup_{(b_1, b_2) \in B} p_1 \{b_1\} \cap p_2 \{b_2\}$ in $\langle Y', p \rangle$
$\text{ap}_{\text{pre}} : (X \xrightarrow{\text{pre}} Y) \Rightarrow \text{Set } Y \Rightarrow \text{Set } X$	$(\circ_{\text{pre}}) : (Y \xrightarrow{\text{pre}} Z) \Rightarrow (X \xrightarrow{\text{pre}} Y) \Rightarrow (X \xrightarrow{\text{pre}} Z)$
$\text{ap}_{\text{pre}} \langle Y', p \rangle B := p(B \cap Y')$	$\langle Z', p_2 \rangle \circ_{\text{pre}} h_1 := \langle Z', \lambda C. \text{ap}_{\text{pre}} h_1 (p_2 C) \rangle$
$\text{domain}_{\text{pre}} : (X \xrightarrow{\text{pre}} Y) \Rightarrow \text{Set } X$	$(\uplus_{\text{pre}}) : (X \xrightarrow{\text{pre}} Y) \Rightarrow (X \xrightarrow{\text{pre}} Y) \Rightarrow (X \xrightarrow{\text{pre}} Y)$
$\text{domain}_{\text{pre}} \langle Y', p \rangle := p Y'$	$h_1 \uplus_{\text{pre}} h_2 := \text{let } Y' := \text{range}_{\text{pre}} h_1 \cup \text{range}_{\text{pre}} h_2$ $p := \lambda B. \text{ap}_{\text{pre}} h_1 B \uplus \text{ap}_{\text{pre}} h_2 B$ in $\langle Y', p \rangle$
$\text{range}_{\text{pre}} : (X \xrightarrow{\text{pre}} Y) \Rightarrow \text{Set } Y$	
$\text{range}_{\text{pre}} \langle Y', p \rangle := Y'$	

Figure 7.5: Lazy preimage mappings and operations.

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

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

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

$$\begin{aligned} \text{ap}_{\text{pre}} &: (X \xrightarrow{\text{pre}} Y) \Rightarrow \text{Set } Y \Rightarrow \text{Set } X \\ \text{ap}_{\text{pre}} \langle Y', p \rangle B &:= p(B \cap Y') \end{aligned} \tag{7.33}$$

Preimage arrow correctness depends on this fact: that using `appre` to compute preimages is the same as computing them from a mapping using `preimage`.

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

Theorem 7.16 (ap_{pre} computes preimages). *Let $g : X \multimap Y$. For all $B \subseteq Y$, $\text{ap}_{\text{pre}} (\text{pre } g) B =$*

preimage g B .

Proof. Expand definitions and apply Lemma 7.15 with $Y' = \text{range } g$. \square

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). $h_1 : X \xrightarrow{\text{pre}} Y$ and $h_2 : X \xrightarrow{\text{pre}} Y$ are equivalent, or $h_1 \equiv h_2$, when $\text{ap}_{\text{pre}} h_1 B \equiv \text{ap}_{\text{pre}} h_2 B$ for all $B \subseteq Y$.

Similarly to proving arrows correct, we prove the operations in Figure 7.5 are correct by proving that pre is a homomorphism (though not an arrow homomorphism): it distributes over mapping operations to yield preimage mapping operations. The remainder of this section states these distributive properties as theorems and proves them. We will use these theorems to derive the preimage arrow from the mapping arrow.

7.5.1 Composition

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

Lemma 7.18 (preimage distributes over (\circ_{map})). Let $g_1 : X \multimap Y$ and $g_2 : Y \multimap Z$. For all $C \subseteq Z$, $\text{preimage} (g_2 \circ_{\text{map}} g_1) C = \text{preimage } g_1 (\text{preimage } g_2 C)$.

Theorem 7.19 (pre distributes over (\circ_{map})). Let $g_1 : X \multimap Y$ and $g_2 : Y \multimap Z$. Then $\text{pre } (g_2 \circ_{\text{map}} g_1) \equiv (\text{pre } g_2) \circ_{\text{pre}} (\text{pre } g_1)$.

Proof. Let $\langle Z', p_2 \rangle := \text{pre } g_2$ and $C \subseteq Z$. Starting from the right-hand side of the equivalence,

$$\begin{aligned} & \text{ap}_{\text{pre}} ((\text{pre } g_2) \circ_{\text{pre}} (\text{pre } g_1)) C \\ & \equiv \text{let } p := \lambda C. \text{ap}_{\text{pre}} (\text{pre } g_1) (p_2 C) \\ & \quad \text{in } p(C \cap Z') \end{aligned} \tag{7.34}$$

Def of ap_{pre} , (\circ_{pre})

$\equiv \text{ap}_{\text{pre}} (\text{pre } g_1) (p_2 (C \cap Z'))$	Def of p
$\equiv \text{ap}_{\text{pre}} (\text{pre } g_1) (\text{ap}_{\text{pre}} (\text{pre } g_2) C)$	Rewrite with ap_{pre}
$\equiv \text{preimage } g_1 (\text{preimage } g_2 C)$	Theorem 7.16
$\equiv \text{preimage } (g_2 \circ_{\text{map}} g_1) C$	Lemma 7.18
$\equiv \text{ap}_{\text{pre}} (\text{pre } (g_2 \circ_{\text{map}} g_1)) C$	Theorem 7.16 □

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 $\langle \cdot, \cdot \rangle_{\text{map}}$ and (\times)). *Let $g_1 : X \rightarrow Y_1$ and $g_2 : X \rightarrow Y_2$. For all $B_1 \subseteq Y_1$ and $B_2 \subseteq Y_2$, $\text{preimage } \langle g_1, g_2 \rangle_{\text{map}} (B_1 \times B_2) = (\text{preimage } g_1 B_1) \cap (\text{preimage } g_2 B_2)$.*

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

$$\bigcup_{j \in J} \text{preimage } g (B_j) = \text{preimage } g \bigcup_{j \in J} B_j \quad (7.35)$$

Theorem 7.22 (`pre` distributes over $\langle \cdot, \cdot \rangle_{\text{map}}$). *Let $g_1 : X \rightarrow Y_1$ and $g_2 : X \rightarrow Y_2$. Then $\text{pre } \langle g_1, g_2 \rangle_{\text{map}} \equiv \langle \text{pre } g_1, \text{pre } g_2 \rangle_{\text{pre}}$.*

Proof. Let $\langle Y'_1, p_1 \rangle := \text{pre } g_1$, $\langle Y'_2, p_2 \rangle := \text{pre } g_2$ and $B \subseteq Y_1 \times Y_2$. Starting from the right-hand side of the equivalence,

$$\begin{aligned} & \text{ap}_{\text{pre}} \langle \text{pre } g_1, \text{pre } g_2 \rangle_{\text{pre}} B \\ & \equiv \text{let } p := \lambda B. \bigcup_{(y_1, y_2) \in B} p_1 \{y_1\} \cap p_2 \{y_2\} && \text{Def of } \text{ap}_{\text{pre}}, \langle \cdot, \cdot \rangle_{\text{pre}} \\ & \quad \text{in } p (B \cap (Y'_1 \times Y'_2)) \end{aligned} \quad (7.36)$$

$$\begin{aligned}
&\equiv \bigcup_{(y_1, y_2) \in B \cap (Y'_1 \times Y'_2)} p_1 \{y_1\} \cap p_2 \{y_2\} && \text{Def of } p \\
&\equiv \bigcup_{(y_1, y_2) \in B \cap (Y'_1 \times Y'_2)} \text{preimage } g_1 \{y_1\} \cap \text{preimage } g_2 \{y_2\} && \text{Theorem 7.16} \\
&\equiv \bigcup_{(y_1, y_2) \in B \cap (Y'_1 \times Y'_2)} \text{preimage } \langle g_1, g_2 \rangle_{\text{map}} (\{y_1\} \times \{y_2\}) && \text{Lemma 7.20} \\
&\equiv \bigcup_{(y_1, y_2) \in B \cap (Y'_1 \times Y'_2)} \text{preimage } \langle g_1, g_2 \rangle_{\text{map}} \{\langle y_1, y_2 \rangle\} && \text{Def of } (\times) \\
&\equiv \text{preimage } \langle g_1, g_2 \rangle_{\text{map}} (B \cap (Y'_1 \times Y'_2)) && \text{Lemma 7.21} \\
&\equiv \text{preimage } \langle g_1, g_2 \rangle_{\text{map}} B && \text{Lemma 7.15} \\
&\equiv \text{ap}_{\text{pre}} (\text{pre } \langle g_1, g_2 \rangle_{\text{map}}) B && \text{Theorem 7.16}
\end{aligned}$$

We have an unmet proof obligation from using Lemma 7.15: that $\text{range } \langle g_1, g_2 \rangle_{\text{map}} \subseteq Y'_1 \times Y'_2$.

Let $b \in \text{range } \langle g_1, g_2 \rangle_{\text{map}}$. By definition of $\langle \cdot, \cdot \rangle_{\text{map}}$, there exists $a \in \text{domain } g_1 \cap \text{domain } g_2$ such that $b = \langle g_1 a, g_2 a \rangle$. Thus, $b \in Y'_1 \times Y'_2$ if and only if $g_1 a \in Y'_1$ and $g_2 a \in Y'_2$.

By definition of pre , $Y'_1 = \text{range } g_1$ and $Y'_2 = \text{range } g_2$. Because $a \in \text{domain } g_1$, $g_1 a \in \text{range } g_1 = Y'_1$. Because $a \in \text{domain } g_2$, $g_2 a \in \text{range } g_2 = Y'_2$. \square

7.5.3 Disjoint Union

Like proving pre distributes over composition, the proof that it distributes over disjoint union simply lifts a lemma about preimage to lazy preimage mappings.

Lemma 7.23 (preimage distributes over (\uplus_{map})). *Let $g_1 : X \rightarrow Y$ and $g_2 : X \rightarrow Y$ have disjoint domains. For all $B \subseteq Y$, $\text{preimage } (g_1 \uplus_{\text{map}} g_2) B = (\text{preimage } g_1 B) \uplus (\text{preimage } g_2 B)$.*

Theorem 7.24 (pre distributes over (\uplus_{map})). *Let $g_1 : X \rightarrow Y$ and $g_2 : X \rightarrow Y$ have disjoint domains. Then $\text{pre } (g_1 \uplus_{\text{map}} g_2) \equiv (\text{pre } g_1) \uplus_{\text{pre}} (\text{pre } g_2)$.*

Proof. Let $Y'_1 := \text{range } g_1$, $Y'_2 := \text{range } g_2$ and $B \subseteq Y$. Starting from the right-hand side of

the equivalence,

$$\begin{aligned}
& \text{ap}_{\text{pre}} ((\text{pre } g_1) \uplus_{\text{pre}} (\text{pre } g_2)) B && (7.37) \\
&\equiv \text{let } p := \lambda B. \text{ap}_{\text{pre}} (\text{pre } g_1) B \uplus \text{ap}_{\text{pre}} (\text{pre } g_2) B && \text{Def of } \text{ap}_{\text{pre}}, (\uplus_{\text{pre}}) \\
&\quad \text{in } p (B \cap (Y'_1 \cup Y'_2)) \\
&\equiv \text{ap}_{\text{pre}} (\text{pre } g_1) (B \cap (Y'_1 \cup Y'_2)) \uplus \text{ap}_{\text{pre}} (\text{pre } g_2) (B \cap (Y'_1 \cup Y'_2)) && \text{Def of } p \\
&\equiv \text{preimage } g_1 (B \cap (Y'_1 \cup Y'_2)) \uplus \text{preimage } g_2 (B \cap (Y'_1 \cup Y'_2)) && \text{Theorem 7.16} \\
&\equiv \text{preimage } (g_1 \uplus_{\text{map}} g_2) (B \cap (Y'_1 \cup Y'_2)) && \text{Lemma 7.23} \\
&\equiv \text{preimage } (g_1 \uplus_{\text{map}} g_2) B && \text{Lemma 7.15} \\
&\equiv \text{ap}_{\text{pre}} (\text{pre } (g_1 \uplus_{\text{map}} g_2)) B && \text{Theorem 7.16}
\end{aligned}$$

We have an unmet proof obligation from using Lemma 7.15: that $\text{range } (g_1 \uplus_{\text{map}} g_2) \subseteq Y'_1 \cup Y'_2$.

Let $b \in \text{range } (g_1 \uplus_{\text{map}} g_2)$. By definition of (\uplus_{map}) , there exists $a \in \text{domain } g_1 \uplus \text{domain } g_2$ such that if $a \in \text{domain } g_1$ then $b = g_1 a$ so $b \in \text{range } g_1 = Y'_1$, and if $a \in \text{domain } g_2$ then $b = g_2 a$ so $b \in \text{range } g_2 = Y'_2$. Thus $b \in Y'_1 \cup Y'_2$. \square

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 $X \rightsquigarrow_{\text{pre}} Y ::= X \rightarrowtail_{\text{pre}} Y$: we run into trouble trying to define arr_{pre} because a preimage mapping needs an observable range. To get one, it is easiest to parameterize preimage computations on a **Set** X ; therefore the *preimage arrow* type constructor is

$$X \rightsquigarrow_{\text{pre}} Y ::= \text{Set } X \Rightarrow (X \rightarrowtail_{\text{pre}} Y) \quad (7.38)$$

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

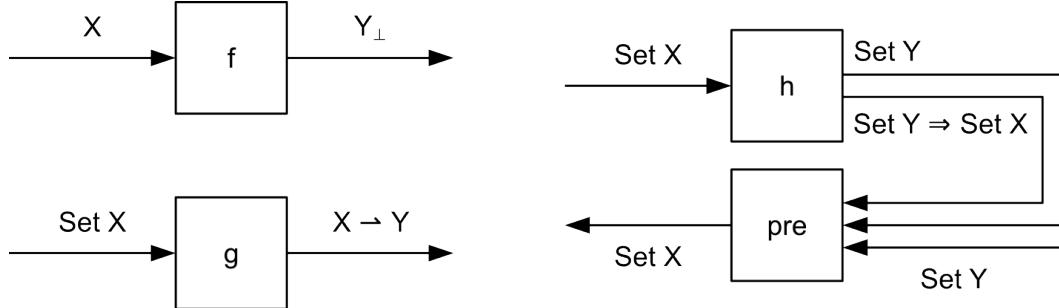


Figure 7.6: Comparison of arrows used as target categories. Computations $f : X \rightsquigarrow_{\perp} Y$ may return an error value \perp . Computations $g : X \rightsquigarrow_{\text{map}} Y$ produce partial mappings on a given $A \subseteq X$, leaving out inputs for which f would return an error. Computations $h : X \rightsquigarrow_{\text{pre}} Y$ produce lazy preimage mappings; i.e. $h A$ computes preimages under $g A$.

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

$$\begin{aligned} \text{lift}_{\text{pre}} : (X \rightsquigarrow_{\text{map}} Y) &\Rightarrow (X \rightsquigarrow_{\text{pre}} Y) \\ \text{lift}_{\text{pre}} g A &:= \text{pre}(g A) \end{aligned} \tag{7.39}$$

By definition of lift_{pre} and Theorem 7.16, for all $g : X \rightsquigarrow_{\text{map}} Y$, and $A \subseteq X$ and $B \subseteq Y$,

$$\begin{aligned} \text{ap}_{\text{pre}}(\text{lift}_{\text{pre}} g A) B &\equiv \text{ap}_{\text{pre}}(\text{pre}(g A)) B \\ &\equiv \text{preimage}(g A) B \end{aligned} \tag{7.40}$$

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 lift_{pre} a homomorphism, we need preimage arrow equivalence to mean “computes the same preimages.”

Definition 7.25 (preimage arrow equivalence). *Two computations $h_1 : X \rightsquigarrow_{\text{pre}} Y$ and $h_2 : X \rightsquigarrow_{\text{pre}} Y$ are equivalent, or $h_1 \equiv h_2$, when $h_1 A \equiv h_2 A$ for all $A \subseteq X$.*

As with arr_{map} , defining arr_{pre} as a composition meets (7.8). The remainder of this section derives $(\&\&_{\text{pre}})$, (\ggg_{pre}) , ifte_{pre} and lazy_{pre} from mapping arrow combinators, in a way that ensures lift_{pre} is an arrow homomorphism from the mapping arrow to the preimage arrow.

$X_{\text{pre}} \rightsquigarrow Y := \text{Set } X \Rightarrow (X_{\text{pre}} \rightarrow Y)$ $\text{arr}_{\text{pre}} : (X \Rightarrow Y) \Rightarrow (X_{\text{pre}} \rightsquigarrow Y)$ $\text{arr}_{\text{pre}} := \text{lift}_{\text{pre}} \circ \text{arr}_{\text{map}}$ $(\ggg_{\text{pre}}) : (X_{\text{pre}} \rightsquigarrow Y) \Rightarrow (Y_{\text{pre}} \rightsquigarrow Z) \Rightarrow (X_{\text{pre}} \rightsquigarrow Z)$ $(h_1 \ggg_{\text{pre}} h_2) A := \text{let } h'_1 := h_1 A$ $\quad \quad \quad h'_2 := h_2 (\text{range}_{\text{pre}} h'_1)$ $\quad \quad \quad \text{in } h'_2 \circ_{\text{pre}} h'_1$ $(\&&_{\text{pre}}) : (X_{\text{pre}} \rightsquigarrow Y) \Rightarrow (X_{\text{pre}} \rightsquigarrow Z) \Rightarrow (X_{\text{pre}} \rightsquigarrow Y \times Z)$ $(h_1 \&&_{\text{pre}} h_2) A := \langle h_1 A, h_2 A \rangle_{\text{pre}}$	$\text{ifte}_{\text{pre}} : (X_{\text{pre}} \rightsquigarrow \text{Bool}) \Rightarrow (X_{\text{pre}} \rightsquigarrow Y) \Rightarrow (X_{\text{pre}} \rightsquigarrow Y)$ $\text{ifte}_{\text{pre}} h_1 h_2 h_3 A := \text{let } h'_1 := h_1 A$ $\quad \quad \quad h'_2 := h_2 (\text{ap}_{\text{pre}} h'_1 \{\text{true}\})$ $\quad \quad \quad h'_3 := h_3 (\text{ap}_{\text{pre}} h'_1 \{\text{false}\})$ $\quad \quad \quad \text{in } h'_2 \uplus_{\text{pre}} h'_3$ $\text{lazy}_{\text{pre}} : (1 \Rightarrow (X_{\text{pre}} \rightsquigarrow Y)) \Rightarrow (X_{\text{pre}} \rightsquigarrow Y)$ $\text{lazy}_{\text{pre}} h A := \text{if } (A = \emptyset) (\text{pre } \emptyset) (h 0 A)$
---	--

Figure 7.7: Preimage arrow definitions.

Figure 7.7 contains the resulting definitions.

7.6.1 Composition

Starting with the left-hand side of (7.9),

$$\begin{aligned}
 & \text{ap}_{\text{pre}} (\text{lift}_{\text{pre}} (g_1 \ggg_{\text{map}} g_2) A) C & (7.41) \\
 & \equiv \text{let } g'_1 := g_1 A & \text{Def of lift}_{\text{pre}}, (\ggg_{\text{map}}) \\
 & \quad g'_2 := g_2 (\text{range } g'_1) \\
 & \quad \text{in } \text{ap}_{\text{pre}} (\text{pre} (g'_2 \circ_{\text{map}} g'_1)) C \\
 & \equiv \text{let } g'_1 := g_1 A & \text{Theorem 7.19} \\
 & \quad g'_2 := g_2 (\text{range } g'_1) \\
 & \quad \text{in } \text{ap}_{\text{pre}} ((\text{pre } g'_1) \circ_{\text{pre}} (\text{pre } g'_2)) C \\
 & \equiv \text{let } h_1 := \text{lift}_{\text{pre}} g_1 A & \text{Rewrite with lift}_{\text{pre}} \\
 & \quad h_2 := \text{lift}_{\text{pre}} g_2 (\text{range}_{\text{pre}} h_1) \\
 & \quad \text{in } \text{ap}_{\text{pre}} (h_2 \circ_{\text{pre}} h_1) C
 \end{aligned}$$

Substituting h_1 for $\text{lift}_{\text{pre}} g_1$ and h_2 for $\text{lift}_{\text{pre}} g_2$, and removing the application of ap_{pre} from both sides of the equivalence gives a definition of (\ggg_{pre}) (Figure 7.7) for which (7.9) holds.

7.6.2 Pairing

Starting with the left-hand side of (7.10),

$$\text{ap}_{\text{pre}} (\text{lift}_{\text{pre}} (\text{g}_1 \&\&_{\text{map}} \text{g}_2) A) B \quad (7.42)$$

$$\equiv \text{ap}_{\text{pre}} (\text{pre} \langle \text{g}_1 A, \text{g}_2 A \rangle_{\text{map}}) B \quad \text{Def of lift}_{\text{pre}}, (\&\&_{\text{map}})$$

$$\equiv \text{ap}_{\text{pre}} \langle \text{pre} (\text{g}_1 A), \text{pre} (\text{g}_2 A) \rangle_{\text{pre}} B \quad \text{Theorem 7.22}$$

$$\equiv \text{ap}_{\text{pre}} \langle \text{lift}_{\text{pre}} \text{g}_1 A, \text{lift}_{\text{pre}} \text{g}_2 A \rangle_{\text{pre}} B \quad \text{Rewrite with lift}_{\text{pre}}$$

Substituting h_1 for $\text{lift}_{\text{pre}} \text{g}_1$ and h_2 for $\text{lift}_{\text{pre}} \text{g}_2$, and removing the application of ap_{pre} from both sides of the equivalence gives a definition of $(\&\&_{\text{pre}})$ (Figure 7.7) for which (7.10) holds.

7.6.3 Conditional

Starting with the left-hand side of (7.11),

$$\text{ap}_{\text{pre}} (\text{lift}_{\text{pre}} (\text{ifte}_{\text{map}} \text{g}_1 \text{g}_2 \text{g}_3) A) B \quad (7.43)$$

$$\equiv \text{let } g'_1 := \text{g}_1 A \\ g'_2 := \text{g}_2 (\text{preimage } g'_1 \{\text{true}\}) \\ g'_3 := \text{g}_3 (\text{preimage } g'_1 \{\text{false}\}) \\ \text{in } \text{ap}_{\text{pre}} (\text{pre} (g'_2 \uplus_{\text{map}} g'_3)) B \quad \text{Def of lift}_{\text{pre}}, \text{ ifte}_{\text{map}}$$

$$\equiv \text{let } g'_1 := \text{g}_1 A \\ g'_2 := \text{g}_2 (\text{preimage } g'_1 \{\text{true}\}) \\ g'_3 := \text{g}_3 (\text{preimage } g'_1 \{\text{false}\}) \\ \text{in } \text{ap}_{\text{pre}} ((\text{pre } g'_2) \uplus_{\text{pre}} (\text{pre } g'_3)) B \quad \text{Theorem 7.24}$$

$$\equiv \text{let } g'_1 := \text{g}_1 A \\ g'_2 := \text{g}_2 (\text{ap}_{\text{pre}} (\text{pre } g'_1) \{\text{true}\}) \\ g'_3 := \text{g}_3 (\text{ap}_{\text{pre}} (\text{pre } g'_1) \{\text{false}\}) \\ \text{in } \text{ap}_{\text{pre}} ((\text{pre } g'_2) \uplus_{\text{pre}} (\text{pre } g'_3)) B \quad \text{Theorem 7.16}$$

$$\equiv \text{let } h_1 := \text{lift}_{\text{pre}} \text{g}_1 A \\ h_2 := \text{lift}_{\text{pre}} \text{g}_2 (\text{ap}_{\text{pre}} h_1 \{\text{true}\}) \\ h_3 := \text{lift}_{\text{pre}} \text{g}_3 (\text{ap}_{\text{pre}} h_1 \{\text{false}\}) \\ \text{in } \text{ap}_{\text{pre}} (h_2 \uplus_{\text{pre}} h_3) B \quad \text{Rewrite with lift}_{\text{pre}}$$

Substituting h_1 , h_2 and h_3 for $\text{lift}_{\text{pre}} g_1$, $\text{lift}_{\text{pre}} g_2$ and $\text{lift}_{\text{pre}} g_3$, and removing the application of ap_{pre} from both sides of the equivalence gives a definition of ifte_{pre} (Figure 7.7) for which (7.11) holds.

7.6.4 Laziness

Starting with the left-hand side of (7.12),

$$\begin{aligned}
 & \text{ap}_{\text{pre}} (\text{lift}_{\text{pre}} (\text{lazy}_{\text{map}} g) A) B \\
 & \equiv \text{let } g' := \text{if } (A = \emptyset) \text{ } \emptyset \text{ } (\text{g } 0 A) \\
 & \quad \text{in } \text{ap}_{\text{pre}} (\text{pre } g') B && \text{Def of } \text{lift}_{\text{pre}}, \text{ lazy}_{\text{map}} \\
 & \equiv \text{let } h := \text{if } (A = \emptyset) \text{ } (\text{pre } \emptyset) \text{ } (\text{pre } (\text{g } 0 A)) \\
 & \quad \text{in } \text{ap}_{\text{pre}} h B && \text{Dist pre over if} \\
 & \equiv \text{let } h := \text{if } (A = \emptyset) \text{ } (\text{pre } \emptyset) \text{ } (\text{lift}_{\text{pre}} (\text{g } 0) A) \\
 & \quad \text{in } \text{ap}_{\text{pre}} h B && \text{Rewrite with } \text{lift}_{\text{pre}}
 \end{aligned} \tag{7.44}$$

Substituting $h 0$ for $\text{lift}_{\text{pre}} (g 0)$ and removing the application of ap_{pre} from both sides of the equivalence gives a definition for lazy_{pre} (Figure 7.7) for which (7.12) holds.

7.6.5 Correctness

Theorem 7.26 (preimage arrow correctness). *lift_{pre} is a homomorphism.*

Proof. By construction. □

Corollary 7.27 (semantic correctness). *For all e , $\llbracket e \rrbracket_{\text{pre}} \equiv \text{lift}_{\text{pre}} \llbracket e \rrbracket_{\text{map}}$.*

As with the mapping arrow, preimage arrow computations can be unruly. We would like to assume that each $h : X \rightsquigarrow_{\text{pre}} Y$ 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 $h : X \rightsquigarrow_{\text{pre}} Y$. If there exists a $g : X \rightsquigarrow_{\text{map}} Y$ such that $h \equiv \text{lift}_{\text{pre}} g$, then h obeys the **preimage arrow law**.*

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

Theorem 7.29. lift_{pre} is an arrow epimorphism.

Proof. Follows from Theorem 7.26 and restriction of $X \rightsquigarrow_{\text{pre}} Y$ to instances for which the preimage arrow law (Definition 7.28) holds. \square

Corollary 7.30. arr_{pre} , $(\&\&_{\text{pre}})$, $(\gg\gg_{\text{pre}})$, ifte_{pre} and lazy_{pre} define an arrow.

7.7 Preimages Under Partial, Probabilistic Functions

We have defined everything on the top of our roadmap:

$$\begin{array}{ccccc}
 X \rightsquigarrow_{\perp} Y & \xrightarrow{\text{lift}_{\text{map}}} & X \rightsquigarrow_{\text{map}} Y & \xrightarrow{\text{lift}_{\text{pre}}} & X \rightsquigarrow_{\text{pre}} Y \\
 \eta_{\perp^*} \downarrow & & \downarrow \eta_{\text{map}^*} & & \downarrow \eta_{\text{pre}^*} \\
 X \rightsquigarrow_{\perp^*} Y & \xrightarrow{\text{lift}_{\text{map}^*}} & X \rightsquigarrow_{\text{map}^*} Y & \xrightarrow{\text{lift}_{\text{pre}^*}} & X \rightsquigarrow_{\text{pre}^*} Y
 \end{array} \tag{7.45}$$

and proved that lift_{map} and lift_{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 $\llbracket e \rrbracket_{\perp} : X \rightsquigarrow_{\perp} Y$, an intensional function that may raise errors.
2. As $\llbracket e \rrbracket_{\text{map}} : X \rightsquigarrow_{\text{map}} Y$, which produces mappings, or extensional functions, on a restricted domain (correct by homomorphism of lift_{map}).
3. As $\llbracket e \rrbracket_{\text{pre}} : X \rightsquigarrow_{\text{pre}} Y$, which computes preimages under mappings produced by $\llbracket e \rrbracket_{\text{map}}$ (correct by homomorphism of lift_{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 $\llbracket e \rrbracket_{\perp}$ does not terminate on just one input in $A \subseteq X$, neither $\llbracket e \rrbracket_{\text{map}} A$ nor $\llbracket e \rrbracket_{\text{pre}} A$ 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 `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 `random < p`:

$$\text{geometric } p := \text{if } (\text{random} < p) \ 0 \ (1 + \text{geometric } p) \quad (7.46)$$

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

Suppose we interpret `geometric p` as $h : \Omega \xrightarrow{\text{pre}} \mathbb{N}$, a preimage arrow computation from random sources $\omega \in \Omega$ to naturals, and we have a probability measure $P : \text{Set} \Omega \rightharpoonup [0, 1]$. The probability of $N \subseteq \mathbb{N}$ is then $P(\text{ap}_{\text{pre}}(h \Omega) N)$. To compute this, we must

- Ensure $\text{ap}_{\text{pre}}(h \Omega) N$ terminates.
- Ensure each $\omega \in \Omega$ contains enough random numbers.
- Determine how `random` indexes numbers in ω .

Ensuring $\text{ap}_{\text{pre}}(h \Omega) N$ 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 $\omega \in \Omega$ through computations, ω is made an infinite stream. Each computation receives and returns an ω . The interpretation of `random` as a computation takes ω ’s head and returns its tail. Combinators pass ω unchanged to one subcomputation, and pass the resulting ω' 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 ω an infinite binary tree. Each computation receives an ω but does not return one. The interpretation of `random` as a computation simply returns ω 's root value. Combinators ignore the root, split ω into a left subtree ω_{left} and a right subtree ω_{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 J be an index set, $j_0 \in J$ a distinguished element, and $\text{left} : J \Rightarrow J$ and $\text{right} : J \Rightarrow J$ be total, injective functions. If for all $j \in J$, $j = \text{next } j_0$ for some finite composition `next` of `left` and `right`, then J , j_0 , `left` and `right` define a **binary indexing scheme**.*

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

$$\begin{aligned}\text{left } (p/q) &:= (p - \frac{1}{2}) / q \\ \text{right } (p/q) &:= (p + \frac{1}{2}) / q\end{aligned}\tag{7.47}$$

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

In any case, J is countable, and can be thought of as a set of indexes into an infinite binary tree. Values of type $J \rightarrow A$ encode such trees of values in A as total mappings (i.e. infinite vectors).

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

$x \rightsquigarrow_{a^*} y ::= \text{AStore } s (x \rightsquigarrow_a y) ::= J \Rightarrow (\langle s, x \rangle \rightsquigarrow_a y)$	$\text{ifte}_{a^*} : (x \rightsquigarrow_{a^*} \text{Bool}) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y)$
$\text{arr}_{a^*} : (x \Rightarrow y) \Rightarrow (x \rightsquigarrow_{a^*} y)$	$\text{ifte}_{a^*} k_1 k_2 k_3 j := \text{ifte}_a (k_1 (\text{left } j))$
$\text{arr}_{a^*} := \eta_{a^*} \circ \text{arr}_a$	$(k_2 (\text{left } (\text{right } j)))$
	$(k_3 (\text{right } (\text{right } j)))$
$(\ggg_{a^*}) : (x \rightsquigarrow_{a^*} y) \Rightarrow (y \rightsquigarrow_{a^*} z) \Rightarrow (x \rightsquigarrow_{a^*} z)$	$\text{lazy}_{a^*} : (1 \Rightarrow (x \rightsquigarrow_{a^*} y)) \Rightarrow (x \rightsquigarrow_{a^*} y)$
$(k_1 \ggg_{a^*} k_2) j :=$	$\text{lazy}_{a^*} k j := \text{lazy}_a \lambda 0. k 0 j$
$(\text{arr}_a \text{ fst } \&&_a k_1 (\text{left } j)) \ggg_a k_2 (\text{right } j)$	<hr/>
$(\&&_{a^*}) : (x \rightsquigarrow_{a^*} y_1) \Rightarrow (x \rightsquigarrow_{a^*} y_2) \Rightarrow (x \rightsquigarrow_{a^*} \langle y_1, y_2 \rangle)$	$\eta_{a^*} : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_{a^*} y)$
$(k_1 \&&_{a^*} k_2) j := k_1 (\text{left } j) \&&_a k_2 (\text{right } j)$	$\eta_{a^*} f j := \text{arr}_a \text{ snd } \ggg_a f$

Figure 7.8: AStore (associative store) arrow transformer definitions.

arrow type, and combinators for arrows of the produced type. The applicative store arrow transformer's type constructor takes a store type s and an arrow type $x \rightsquigarrow_a y$:

$$\text{AStore } s (x \rightsquigarrow_a y) ::= J \Rightarrow (\langle s, x \rangle \rightsquigarrow_a y) \quad (7.48)$$

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

$$\begin{aligned} \eta_{a^*} &: (x \rightsquigarrow_a y) \Rightarrow \text{AStore } s (x \rightsquigarrow_a y) \\ \eta_{a^*} f j &:= \text{arr}_a \text{ snd } \ggg_a f \end{aligned} \quad (7.49)$$

Figure 7.8 defines the remaining combinators. Each subcomputation receives $\text{left } j$, $\text{right } j$, 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.

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 $\Omega := J \rightarrow [0, 1]$. A **random source** is any infinite*

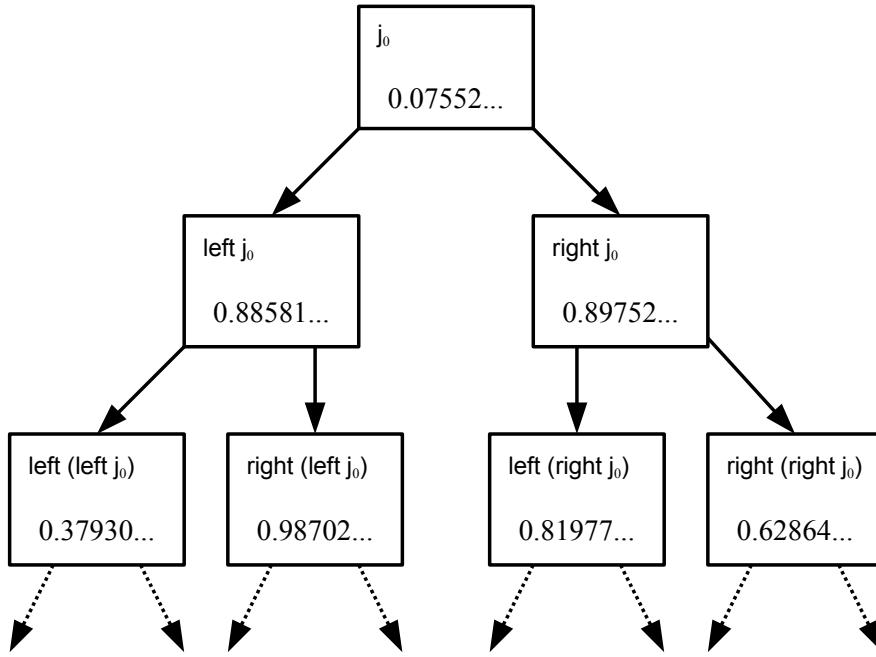


Figure 7.9: An $\omega \in \Omega$ is an infinite binary tree of random values encoded as a total mapping from tree indexes in J to real numbers in $[0, 1]$.

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 $t : J \rightarrow \text{Bool}_\perp$ such that $t j = \text{true}$ or $t j = \text{false}$ for no more than finitely many $j \in J$.

Let $T \subset J \rightarrow \text{Bool}_\perp$ be the largest set of branch traces.

Now $X \rightsquigarrow_{a^*} Y ::= \text{AStore } (\Omega \times T) (X \rightsquigarrow_a Y)$ is an AStore arrow type whose computations thread both random stores and branch traces.

For probabilistic programs, we define a combinator random_{a^*} that returns the number at its tree index in the random source, and extend $\llbracket \cdot \rrbracket_{a^*}$ for arrows a^* for which random_{a^*} is

defined:

$$\begin{aligned}
 \text{random}_{\mathbf{a}^*} : X \rightsquigarrow_{\mathbf{a}^*} [0, 1] \\
 \text{random}_{\mathbf{a}^*} j := \text{arr}_a (\text{fst} \ggg \text{fst} \ggg \pi j) \\
 \llbracket \text{random} \rrbracket_{\mathbf{a}^*} := \text{random}_{\mathbf{a}^*}
 \end{aligned} \tag{7.50}$$

Here, πj projects its argument onto the argument's j th coordinate:

$$\begin{aligned}
 \pi : J \Rightarrow (J \rightarrow X) \Rightarrow X \\
 \pi j f := f j
 \end{aligned} \tag{7.51}$$

So πj 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:

$$\begin{aligned}
 \text{branch}_{\mathbf{a}^*} : X \rightsquigarrow_{\mathbf{a}^*} \text{Bool} \\
 \text{branch}_{\mathbf{a}^*} j := \text{arr}_a (\text{fst} \ggg \text{snd} \ggg \pi j) \\
 \text{ifte}_{\mathbf{a}^*}^\Downarrow : (x \rightsquigarrow_{\mathbf{a}^*} \text{Bool}) \Rightarrow (x \rightsquigarrow_{\mathbf{a}^*} y) \Rightarrow (x \rightsquigarrow_{\mathbf{a}^*} y) \Rightarrow (x \rightsquigarrow_{\mathbf{a}^*} y) \\
 \text{ifte}_{\mathbf{a}^*}^\Downarrow k_1 k_2 k_3 j := \text{ifte}_a ((k_1 (\text{left } j) \& \&_a \text{branch}_{\mathbf{a}^*} j) \ggg_a \text{arr}_a \text{agrees}) \\
 (k_2 (\text{left } (\text{right } j))) \\
 (k_3 (\text{right } (\text{right } j)))
 \end{aligned} \tag{7.52}$$

where $\text{agrees } \langle b_1, b_2 \rangle := \text{if } (b_1 = b_2) b_1 \perp$. Thus, if the branch trace does not agree with the test expression, it returns an error. We define a new semantic function $\llbracket \cdot \rrbracket_{\mathbf{a}^*}^\Downarrow$ by replacing the if rule in $\llbracket \cdot \rrbracket_{\mathbf{a}^*}$:

$$\llbracket \text{if } e_c e_t e_f \rrbracket_{\mathbf{a}^*}^\Downarrow := \text{ifte}_{\mathbf{a}^*}^\Downarrow \llbracket e_c \rrbracket_{\mathbf{a}^*}^\Downarrow \llbracket \text{lazy } e_t \rrbracket_{\mathbf{a}^*}^\Downarrow \llbracket \text{lazy } e_f \rrbracket_{\mathbf{a}^*}^\Downarrow \tag{7.53}$$

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

Definition 7.34 (terminating, probabilistic arrows). Define

$$\begin{aligned} X \rightsquigarrow_{\perp^*} Y &::= \text{AStore } (\Omega \times T) (X \rightsquigarrow_{\perp} Y) \\ X \rightsquigarrow_{\text{map}^*} Y &::= \text{AStore } (\Omega \times T) (X \rightsquigarrow_{\text{map}} Y) \\ X \rightsquigarrow_{\text{pre}^*} Y &::= \text{AStore } (\Omega \times T) (X \rightsquigarrow_{\text{pre}} Y) \end{aligned} \tag{7.54}$$

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 **AStore** arrow equivalence to be more extensional.

Definition 7.35 (**AStore** arrow equivalence). Two **AStore** arrow computations k_1 and k_2 are equivalent, or $k_1 \equiv k_2$, when $k_1 j \equiv k_2 j$ for all $j \in J$.

Pure Expressions

Proving η_{a^*} is a homomorphism proves $\llbracket \cdot \rrbracket_{a^*}$ correctly interprets pure expressions. Because **AStore** accepts any arrow type $x \rightsquigarrow_a y$, we can do so using only the arrow laws. From here on, we assume every **AStore** arrow's base type's combinators obey the arrow laws listed in Section 7.2.1.

Theorem 7.36 (pure **AStore** arrow correctness). η_{a^*} is a homomorphism.

Proof. Defining arr_{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 $\text{arr}_a \text{ snd}$.

For (7.12), additionally β -reduce within the outer thunk, then use the lazy distributive law (7.17) to extract $\text{arr}_a \text{ snd}$. \square

Corollary 7.37 (pure semantic correctness). For all pure e , $\llbracket e \rrbracket_{a^*} \equiv \eta_{a^*} \llbracket e \rrbracket_a$.

Effectful Expressions

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

$$\begin{aligned} \text{lift}_{b^*} : (x \rightsquigarrow_{a^*} y) &\Rightarrow (x \rightsquigarrow_{b^*} y) \\ \text{lift}_{b^*} f j &:= \text{lift}_b (f j) \end{aligned} \tag{7.55}$$

where $\text{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$. A commutative diagram shows the relationships more clearly:

$$\begin{array}{ccc} x \rightsquigarrow_a y & \xrightarrow{\text{lift}_b} & x \rightsquigarrow_b y \\ \eta_{a^*} \downarrow & & \downarrow \eta_{b^*} \\ x \rightsquigarrow_{a^*} y & \xrightarrow{\text{lift}_{b^*}} & x \rightsquigarrow_{b^*} y \end{array} \tag{7.56}$$

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

Theorem 7.38 (natural transformation). *If lift_b is an arrow homomorphism, then (7.56) commutes.*

Proof. Expand definitions and apply homomorphism laws (7.9) and (7.8) for lift_b :

$$\begin{aligned} \text{lift}_{b^*} (\eta_{a^*} f) &\equiv \lambda j. \text{lift}_b (\text{arr}_a \text{ snd} \ggg_a f) \\ &\equiv \lambda j. \text{lift}_b (\text{arr}_a \text{ snd}) \ggg_b \text{lift}_b f \\ &\equiv \lambda j. \text{arr}_b \text{ snd} \ggg_b \text{lift}_b f \\ &\equiv \eta_{b^*} (\text{lift}_b f) \end{aligned} \tag{7.57} \quad \square$$

Theorem 7.39 (effectful AStore arrow correctness). *If lift_b is an arrow homomorphism from a to b , then lift_{b^*} is an arrow homomorphism from a^* to b^* .*

Proof. For each homomorphism property (7.8)–(7.12), expand the definitions of lift_{b^*} and the combinator, distribute lift_b , rewrite in terms of lift_{b^*} , and rewrite using the definition of the

combinator. For example, for distribution over pairing:

$$\begin{aligned}
\text{lift}_{b^*} (k_1 \&&_{a^*} k_2) j &\equiv \text{lift}_b ((k_1 \&&_{a^*} k_2) j) \\
&\equiv \text{lift}_b (k_1 (\text{left } j) \&&_a k_2 (\text{right } j)) \\
&\equiv \text{lift}_b (k_1 (\text{left } j)) \&&_b \text{lift}_b (k_2 (\text{right } j)) \\
&\equiv (\text{lift}_{b^*} k_1) (\text{left } j) \&&_b (\text{lift}_{b^*} k_2) (\text{right } j) \\
&\equiv (\text{lift}_{b^*} k_1 \&&_{b^*} \text{lift}_{b^*} k_2) j
\end{aligned} \tag{7.58}$$

The remaining properties are similar, though distributing lift_{b^*} over lazy_{a^*} requires defining an extra thunk in the last step. \square

Corollary 7.40 (effectful semantic correctness). *If lift_b is an arrow homomorphism, then for all expressions e , $\llbracket e \rrbracket_{b^*} \equiv \text{lift}_{b^*} \llbracket e \rrbracket_{a^*}$ and $\llbracket e \rrbracket_{b^*}^\downarrow \equiv \text{lift}_{b^*} \llbracket e \rrbracket_{a^*}^\downarrow$.*

Corollary 7.41 (mapping* and preimage* arrow correctness). *The following diagram commutes:*

$$\begin{array}{ccccc}
X \rightsquigarrow_{\perp} Y & \xrightarrow{\text{lift}_{\text{map}}} & X \rightsquigarrow_{\text{map}} Y & \xrightarrow{\text{lift}_{\text{pre}}} & X \rightsquigarrow_{\text{pre}} Y \\
\eta_{\perp^*} \downarrow & & \downarrow \eta_{\text{map}^*} & & \downarrow \eta_{\text{pre}^*} \\
X \rightsquigarrow_{\perp^*} Y & \xrightarrow{\text{lift}_{\text{map}^*}} & X \rightsquigarrow_{\text{map}^*} Y & \xrightarrow{\text{lift}_{\text{pre}^*}} & X \rightsquigarrow_{\text{pre}^*} Y
\end{array} \tag{7.59}$$

Further, $\text{lift}_{\text{map}^*}$ and $\text{lift}_{\text{pre}^*}$ 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 $\text{lift}_{\text{map}^*}$ and $\text{lift}_{\text{pre}^*}$ being arrow homomorphisms, and Theorem 7.6.

Corollary 7.42 (effectful semantic correctness). *For all expressions e ,*

$$\begin{aligned}
\llbracket e \rrbracket_{\text{pre}^*} &\equiv \text{lift}_{\text{pre}^*} (\text{lift}_{\text{map}^*} \llbracket e \rrbracket_{\perp^*}) \\
\llbracket e \rrbracket_{\text{pre}^*}^\downarrow &\equiv \text{lift}_{\text{pre}^*} (\text{lift}_{\text{map}^*} \llbracket e \rrbracket_{\perp^*}^\downarrow)
\end{aligned} \tag{7.60}$$

Unfortunately, because a statement such as “ $k_1 \equiv k_2$ ” implies k_1 terminates if and only if k_2 terminates, we cannot use the same tactics to prove an asymmetric statement such as

“ k_2 terminates with the correct answer whenever k_1 terminates; otherwise returns \perp .” 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 $\llbracket e \rrbracket_{a^*}^\Downarrow$ computations, which are interpreted using $\text{ifte}_{a^*}^\Downarrow$ and should always terminate, with $\llbracket e \rrbracket_{a^*}$ computations, which are interpreted using ifte_{a^*} and may not terminate. To do so, we need to find the largest domain on which $\llbracket e \rrbracket_{a^*}^\Downarrow$ and $\llbracket e \rrbracket_{a^*}$ should agree.

Definition 7.43 (maximal domain). *A computation’s **maximal domain** is the largest A^* for which*

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

The maximal domain of $k : X \rightsquigarrow_{a^*} Y$ is that of $k j_0$.

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

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

$$\begin{aligned} \text{restrict} &: (X \rightarrow Y) \Rightarrow \text{Set } X \Rightarrow (X \rightarrow Y) \\ \text{restrict } g A &:= \lambda a \in (A \cap \text{domain } g). g a \end{aligned} \tag{7.61}$$

Theorem 7.44 (mapping arrow restriction). *Let $g : X \rightsquigarrow_{\text{map}} Y$, and $A^\Downarrow \subseteq X$ be the largest for which $g A^\Downarrow$ terminates. For all $A \subseteq A^\Downarrow$, $g A = \text{restrict}(g A^\Downarrow) A$.*

Proof. By the mapping arrow law (Definition 7.12) there is an $f : X \rightsquigarrow_{\perp} Y$ such that $g \equiv \text{lift}_{\text{map}} f$. Thus,

$$\begin{aligned}
\text{restrict } (g A^{\Downarrow}) A &\equiv \text{restrict } (\text{lift}_{\text{map}} f A^{\Downarrow}) A \\
&\equiv \text{restrict } (\{\langle a, b \rangle \in \text{mapping } f A^{\Downarrow} \mid b \neq \perp\}) A \\
&\equiv \{\langle a, b \rangle \in \text{mapping } f A \mid b \neq \perp\} \\
&\equiv \text{lift}_{\text{map}} f A \\
&\equiv g A
\end{aligned} \tag{7.62}$$

□

Theorem 7.45 (domain closure operators). *If $f : X \rightsquigarrow_{\perp} Y$, $g : X \rightsquigarrow_{\text{map}} Y$ and $h : X \rightsquigarrow_{\text{pre}} Y$, then $\text{domain}_{\perp} f$, $\text{domain} \circ g$, and $\text{domain}_{\text{pre}} \circ h$ are monotone, decreasing, and idempotent in the subdomains on which they terminate.*

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

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

Theorem 7.46. *Let $f := \llbracket e \rrbracket_{\perp^*} : X \rightsquigarrow_{\perp^*} Y$ with maximal domain A^* , and $f' := \llbracket e \rrbracket_{\perp^*}^{\Downarrow}$. For all $\langle \langle \omega, t \rangle, a \rangle \in A^*$, there exists a $T' \subseteq T$ such that*

- If $t' \in T'$ then $f' j_0 \langle \langle \omega, t' \rangle, a \rangle = f j_0 \langle \langle \omega, t \rangle, a \rangle$.
- If $t' \in T \setminus T'$ then $f' j_0 \langle \langle \omega, t' \rangle, a \rangle = \perp$.

Proof. Define T' as the set of all $t' \in J \rightarrow \text{Bool}_{\perp}$ such that $t' j = z$ if the subcomputation with index j is an if whose test returns z . Because $f j_0 \langle \langle \omega, t \rangle, a \rangle$ terminates, $t' j \neq \perp$ for at most finitely many j , so each $t' \in T$.

Let $t' \in T'$. Because the test of every if subcomputation at index j agrees with $t' j$ and f ignores branch traces, $f' j_0 \langle \langle \omega, t' \rangle, a \rangle = f j_0 \langle \langle \omega, t \rangle, a \rangle$.

Let $t' \in T \setminus T'$. There exists an if subexpression with a test that does not agree with t' ; therefore $f' j_0 \langle \langle \omega, t' \rangle, a \rangle = \perp$. \square

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

Definition 7.47 (index prefix/suffix). *A finite $J' \subset J$ is an **index prefix** if $J' = \{j_0\}$ or, for some index prefix J'' and $j \in J''$, $J' = J'' \uplus \{\text{left } j\}$ or $J' = J'' \uplus \{\text{right } j\}$. The corresponding **index suffix** is $J \setminus J'$.*

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

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

Theorem 7.48. *Let $f := \llbracket e \rrbracket_{\perp^*} : X \rightsquigarrow_{\perp^*} Y$ with maximal domain A^* , and $f' := \llbracket e \rrbracket_{\perp^*}^\psi$. For all $a \in ((\Omega \times T) \times X) \setminus A^*$, $f' j_0 a = \perp$.*

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

Suppose $f j_0 a$ terminates. If an if subcomputation's test does not agree with t , then $f' j_0 a = \perp$. If every if's test agrees, $f' j_0 a = f j_0 a = \perp$.

Suppose $f j_0 a$ does not terminate. The set of all indexes j for which $t j \neq \perp$ is contained within an index prefix J' . By hypothesis, there is an if subcomputation at some index j' such that $j' \in J \setminus J'$. Because $t j' = \perp$, $f' j_0 a = \perp$. \square

Corollary 7.49. *For all e , the maximal domain of $\llbracket e \rrbracket_{\perp^*}^\psi$ is a subset of that of $\llbracket e \rrbracket_{\perp^*}$.*

Corollary 7.50. *Let $f' := \llbracket e \rrbracket_{\perp^*}^\psi : X \rightsquigarrow_{\perp^*} Y$ with maximal domain A^* , and $f := \llbracket e \rrbracket_{\perp^*}$. For all $a \in A^*$, $f' j_0 a = f j_0 a$.*

Corollary 7.51 (correct computation everywhere). *Let $\llbracket e \rrbracket_{\perp^*}^\Downarrow : X \rightsquigarrow_{\perp^*} Y$ have maximal domain A^* , and $X' := (\Omega \times T) \times X$. For all $a \in X'$, $A \subseteq X'$ and $B \subseteq Y$,*

$$\begin{aligned}\llbracket e \rrbracket_{\perp^*}^\Downarrow j_0 a &= \text{if } (a \in A^*) (\llbracket e \rrbracket_{\perp^*}^\Downarrow j_0 a) \perp \\ \llbracket e \rrbracket_{\text{map}^*}^\Downarrow j_0 A &= \llbracket e \rrbracket_{\text{map}^*} j_0 (A \cap A^*) \\ \text{ap}_{\text{pre}} (\llbracket e \rrbracket_{\text{pre}^*}^\Downarrow j_0 A) B &= \text{ap}_{\text{pre}} (\llbracket e \rrbracket_{\text{pre}^*} j_0 (A \cap A^*)) B\end{aligned}\tag{7.63}$$

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

7.8 Output Probabilities and Measurability

Typically, for $g : \Omega \rightarrow Y$, the probability of $B \subseteq Y$ is $P(\text{preimage } g B)$, where $P : \text{Set } \Omega \rightarrow [0, 1]$ assigns probabilities to subsets of Ω .

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

Not all preimage sets have sensible measures. Sets that do are called **measurable**. Computing preimages and projecting them onto Ω 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 , $\llbracket e \rrbracket_{\text{map}^*}^\Downarrow$ is measurable.*

Second, projecting a program's preimages onto Ω results in a measurable set.

Theorem 7.53. *If $A \subseteq (\Omega \times T) \times \{\langle \rangle\}$ is measurable, then $\text{image } (\text{fst} \ggg \text{fst}) A$ is measurable.*

The proofs of these theorems are in Appendix A.

$\text{id}_{\text{pre}} A := \langle A, \lambda B. B \rangle$	$\text{proj}_1 : \text{Set } \langle X_1, X_2 \rangle \Rightarrow \text{Set } X_1$
$\text{const}_{\text{pre}} b A := \langle \{b\}, \lambda B. \text{if } (B = \emptyset) \circ A \rangle$	$\text{proj}_1 := \text{image fst}$
$\text{fst}_{\text{pre}} A := \langle \text{proj}_1 A, \text{unproj}_1 A \rangle$	$\text{proj}_2 : \text{Set } \langle X_1, X_2 \rangle \Rightarrow \text{Set } X_2$
$\text{snd}_{\text{pre}} A := \langle \text{proj}_2 A, \text{unproj}_2 A \rangle$	$\text{proj}_2 := \text{image snd}$
$\pi_{\text{pre}} j A := \langle \text{proj } j A, \text{unproj } j A \rangle$	
<hr/>	
$\text{proj} : J \Rightarrow \text{Set } (J \rightarrow X) \Rightarrow \text{Set } X$	$\text{unproj}_1 : \text{Set } \langle X_1, X_2 \rangle \Rightarrow \text{Set } X_1 \Rightarrow \text{Set } \langle X_1, X_2 \rangle$
$\text{proj } j A := \text{image } (\pi j) A$	$\text{unproj}_1 A A_1 := \text{preimage } (\text{mapping fst } A) A_1$
	$\equiv A \cap (A_1 \times \text{proj}_2 A)$
<hr/>	
$\text{unproj} : J \Rightarrow \text{Set } (J \rightarrow X) \Rightarrow \text{Set } X \Rightarrow \text{Set } (J \rightarrow X)$	$\text{unproj}_2 : \text{Set } \langle X_1, X_2 \rangle \Rightarrow \text{Set } X_2 \Rightarrow \text{Set } \langle X_1, X_2 \rangle$
$\text{unproj } j A B := \text{preimage } (\text{mapping } (\pi j) A) B$	$\text{unproj}_2 A A_2 := \text{preimage } (\text{mapping snd } A) A_2$
$\equiv A \cap \prod_{i \in J} \text{if } (j = i) B (\text{proj } j A)$	$\equiv A \cap (\text{proj}_1 A \times A_2)$

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 $\text{preimage } (g A) B$ is uncomputable for most mappings g and uncountable sets A and B no matter how cleverly they are represented. Further, because pre , lift_{pre} and arr_{pre} are ultimately defined in terms of preimage , we cannot implement them.

Fortunately, we need to apply arr_{pre} only to certain functions. Figure 7.1 (which defines $\llbracket \cdot \rrbracket_a$) lifts id , $\text{const } b$, fst and snd . Section 7.7.4, which defines the combinators used to interpret partial, probabilistic programs, lifts πj 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 $\text{arr}_{\text{pre}} \text{id}$, $\text{arr}_{\text{pre}} \text{fst}$, $\text{arr}_{\text{pre}} \text{snd}$, $\text{arr}_{\text{pre}} (\text{const } b)$ and

$\text{arr}_{\text{pre}}(\pi j)$. (We will deal with `agrees` separately.) To implement them, we must model sets in a way that makes $A = \emptyset$ is decidable, and the following representable and finitely computable:

- $A \cap B, \emptyset, \{\text{true}\}, \{\text{false}\}$ and $\{b\}$ for every const b
 - $A_1 \times A_2, \text{proj}_1 A$ and $\text{proj}_2 A$
 - $J \rightarrow X, \text{proj } j A$ and $\text{unproj } j A B$
- (7.64)

Before addressing representation and computability, we need to define families of sets under which these operations are closed.

Definition 7.54 (rectangular family). *Rect X denotes the **rectangular family** of subsets of X . Rect X must contain \emptyset and X , and be closed under finite intersections. Products must satisfy the following rules:*

$$\text{Rect } \langle X_1, X_2 \rangle = (\text{Rect } X_1) \boxtimes (\text{Rect } X_2) \quad (7.65)$$

$$\text{Rect } (J \rightarrow X) = (\text{Rect } X)^{\boxtimes J} \quad (7.66)$$

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

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 := \{A_1 \times A_2 \mid A_1 \in \mathcal{A}_1, A_2 \in \mathcal{A}_2\} \quad (7.67)$$

$$\mathcal{A}^{\boxtimes J} := \bigcup_{J' \subset J \text{ finite}} \left\{ \prod_{j \in J'} A_j \mid A_j \in \mathcal{A}, j \in J' \iff A_j \subset \bigcup \mathcal{A} \right\} \quad (7.68)$$

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

Further, all of the operations in (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 $\text{Rect } \langle X_1, X_2 \rangle$ are modeled by pairs of type $\langle \text{Rect } X_1, \text{Rect } X_2 \rangle$. By (7.68), sets in $\text{Rect } (J \rightarrow X)$ have no more than finitely many projections that are proper subsets of X . They can be modeled by *finite* binary trees, where unrepresented projections are implicitly X . Figure 7.11 illustrates a model of a member of $\text{Rect } (J \rightarrow [0, 1])$; i.e. a rectangular subset of Ω .

The set of branch traces T is nonrectangular, but we can model T subsets by $J \rightarrow \text{Bool}_\perp$

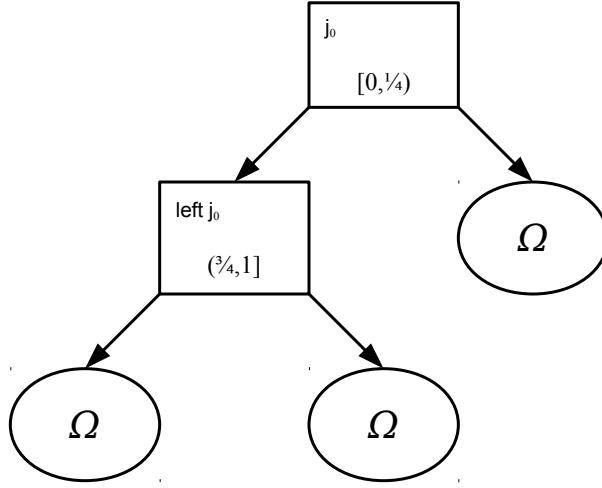


Figure 7.11: A finite binary tree model of $\text{unproj}(\text{left } j_0)(\text{unproj } j_0 \Omega [0, \frac{1}{4})) (\frac{3}{4}, 1]$. Because of Ω 's self-similarity, and because rectangles of $J \rightarrow [0, 1]$ are defined so that only finitely many projections are not $[0, 1]$, every rectangular subset of Ω has a finite binary tree model.

rectangles, implicitly intersected with T .

Theorem 7.55 (T model). *If $T' \in \text{Rect}(J \rightarrow \text{Bool}_\perp)$ and $j \in J$, then $\text{proj } j \ T' = \text{proj } j(T' \cap T)$.*

If $B \subseteq \text{Bool}_\perp$, then $\text{unproj } j(T' \cap T) B = \text{unproj } j T' B \cap T$.

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

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

The statement about unproj is an easy corollary. □

7.9.2 Approximate Preimage Mapping Operations

Implementing lazy_{pre} (defined in Figure 7.7) requires computing pre , but only for the empty mapping, which is trivial: $\text{pre } \emptyset \equiv \langle \emptyset, \lambda B. \emptyset \rangle$. Implementing the other combinators requires (\circ_{pre}) , $\langle \cdot, \cdot \rangle_{\text{pre}}$ and (\cup_{pre}) .

From the preimage mapping definitions (Figure 7.5), we see that ap_{pre} is defined using (\cap) and that (\circ_{pre}) is defined using ap_{pre} , so (\circ_{pre}) is directly implementable. Unfortunately,

we hit a snag with $\langle \cdot, \cdot \rangle_{\text{pre}}$: it loops over possibly uncountably many members of B in a big union. At this point, we need to approximate.

Theorem 7.56 (pair preimage approximation). *Let $g_1 : X \rightarrow Y_1$ and $g_2 : X \rightarrow Y_2$. For all $B \subseteq Y_1 \times Y_2$, preimage $\langle g_1, g_2 \rangle_{\text{map}} B \subseteq \text{preimage } g_1 (\text{proj}_1 B) \cap \text{preimage } g_2 (\text{proj}_2 B)$.*

Proof. By monotonicity of preimages and projections, and by Lemma 7.20. \square

It is not hard to use Theorem 7.56 to show that

$$\begin{aligned} \langle \cdot, \cdot \rangle'_{\text{pre}} : (X \xrightarrow{\text{pre}} Y_1) &\Rightarrow (X \xrightarrow{\text{pre}} Y_2) \Rightarrow (X \xrightarrow{\text{pre}} Y_1 \times Y_2) \\ \langle \langle Y'_1, p_1 \rangle, \langle Y'_2, p_2 \rangle \rangle'_{\text{pre}} &:= \langle Y'_1 \times Y'_2, \lambda B. p_1 (\text{proj}_1 B) \cap p_2 (\text{proj}_2 B) \rangle \end{aligned} \quad (7.69)$$

computes covering rectangles of preimages under pairing.

For (\uplus_{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:

$$\begin{aligned} (A_1 \times A_2) \vee (B_1 \times B_2) &= (A_1 \vee B_1) \times (A_2 \vee B_2) \\ (\prod_{j \in J} A_j) \vee (\prod_{j \in J} B_j) &= \prod_{j \in J} A_j \vee B_j \end{aligned} \quad (7.70)$$

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

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

$$\begin{aligned} (\uplus'_{\text{pre}}) : (X \xrightarrow{\text{pre}} Y) &\Rightarrow (X \xrightarrow{\text{pre}} Y) \Rightarrow (X \xrightarrow{\text{pre}} Y) \\ h_1 \uplus'_{\text{pre}} h_2 &:= \text{let } Y' := \text{range}_{\text{pre}} h_1 \vee \text{range}_{\text{pre}} h_2 \\ &\quad p := \lambda B. \text{ap}_{\text{pre}} h_1 B \vee \text{ap}_{\text{pre}} h_2 B \\ &\quad \text{in } \langle Y', p \rangle \end{aligned} \quad (7.71)$$

To interpret programs that may not terminate, or that terminate with probability 1, we need to approximate $\text{ifte}_{\text{pre}*}^{\Downarrow}$ (7.52), which is defined in terms of `agrees`. Defining its approximation in terms of an approximation of `agrees` would not allow us to preserve the fact that expressions interpreted using $\text{ifte}_{\text{pre}*}^{\Downarrow}$ always terminate. The best approximation of the

preimage of Bool under agrees (as a mapping) is $\text{Bool} \times \text{Bool}$, which contains $\langle \text{true}, \text{false} \rangle$ and $\langle \text{false}, \text{true} \rangle$, and thus would not constrain the test to agree with the branch trace.

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

$$\begin{aligned}
 \text{ifte}_{\text{pre}^*}^\psi k_1 k_2 k_3 j A &\equiv \text{let } \langle C_k, p_k \rangle := k_1 j_1 A \\
 &\quad \langle C_b, p_b \rangle := \text{branch}_{\text{pre}^*} j A \\
 &\quad C_2 := C_k \cap C_b \cap \{\text{true}\} \\
 &\quad C_3 := C_k \cap C_b \cap \{\text{false}\} \\
 &\quad A_2 := p_k C_2 \cap p_b C_2 \\
 &\quad A_3 := p_k C_3 \cap p_b C_3 \\
 &\text{in } k_2 j_2 A_2 \uplus_{\text{pre}} k_3 j_3 A_3
 \end{aligned} \tag{7.72}$$

where $j_1 = \text{left } j$ and so on. Unfortunately, a straightforward approximation of this would still take unnecessary branches, when A_2 or A_3 overapproximates \emptyset .

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

We need an overapproximation for the non-branching case. In the exact semantics, the returned preimage mapping's range is a subset of Y , and it returns subsets of $A_2 \uplus A_3$. Therefore, $\text{ifte}_{\text{pre}^*}^\psi$ may return $\langle Y, \lambda B. A_2 \vee A_3 \rangle$ when $C_b = \{\text{true}, \text{false}\}$. 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 $\text{Rect } Y$.

Figure 7.12b defines the final approximating preimage arrow. This arrow, the lifts in Figure 7.10, and the semantic function $\llbracket \cdot \rrbracket_a$ in Figure 7.1 define an approximating semantics for partial, probabilistic programs.

$$\begin{array}{ll}
X_{\text{pre}}' Y ::= \langle \text{Rect } Y, \text{Rect } Y \Rightarrow \text{Rect } X \rangle & \langle \cdot, \cdot \rangle'_{\text{pre}} : (X_{\text{pre}}' Y_1) \Rightarrow (X_{\text{pre}}' Y_2) \Rightarrow (X_{\text{pre}}' Y_1 \times Y_2) \\
\emptyset'_{\text{pre}} := \langle \emptyset, \lambda B. \emptyset \rangle & \langle \langle Y'_1, p_1 \rangle, \langle Y'_2, p_2 \rangle \rangle'_{\text{pre}} := \\
& (Y'_1 \times Y'_2, \lambda B. p_1 (\text{proj}_1 B) \cap p_2 (\text{proj}_2 B)) \\
\text{ap}'_{\text{pre}} : (X_{\text{pre}}' Y) \Rightarrow \text{Rect } Y \Rightarrow \text{Rect } X & (\sqcup'_{\text{pre}}) : (X_{\text{pre}}' Y) \Rightarrow (X_{\text{pre}}' Y) \Rightarrow (X_{\text{pre}}' Y) \\
\text{ap}'_{\text{pre}} \langle Y', p \rangle B := p (B \cap Y') & \langle Y'_1, p_1 \rangle \sqcup'_{\text{pre}} \langle Y'_2, p_2 \rangle := \\
& \langle Y'_1 \vee Y'_2, \lambda B. \text{ap}'_{\text{pre}} \langle Y'_1, p_1 \rangle B \vee \text{ap}'_{\text{pre}} \langle Y'_2, p_2 \rangle B \rangle \\
(\circ'_{\text{pre}}) : (Y_{\text{pre}}' Z) \Rightarrow (X_{\text{pre}}' Y) \Rightarrow (X_{\text{pre}}' Z) & \\
\langle Z', p_2 \rangle \circ'_{\text{pre}} h_1 := \langle Z', \lambda C. \text{ap}'_{\text{pre}} h_1 (p_2 C) \rangle &
\end{array}$$

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

$$\begin{array}{ll}
X_{\text{pre}}' Y ::= \text{Rect } X \Rightarrow (X_{\text{pre}}' Y) & \text{ifte}'_{\text{pre}} : (X_{\text{pre}}' \text{Bool}) \Rightarrow (X_{\text{pre}}' Y) \Rightarrow (X_{\text{pre}}' Y) \Rightarrow (X_{\text{pre}}' Y) \\
(\ggg'_{\text{pre}}) : (X_{\text{pre}}' Y) \Rightarrow (Y_{\text{pre}}' Z) \Rightarrow (X_{\text{pre}}' Z) & \text{ifte}'_{\text{pre}} h_1 h_2 h_3 A := \text{let } h'_1 := h_1 A \\
(h_1 \ggg'_{\text{pre}} h_2) A := \text{let } h'_1 := h_1 A & \quad h'_2 := h_2 (\text{ap}'_{\text{pre}} h'_1 \{\text{true}\}) \\
& \quad h'_3 := h_3 (\text{ap}'_{\text{pre}} h'_1 \{\text{false}\}) \\
& \quad \text{in } h'_2 \sqcup'_{\text{pre}} h'_3 \\
(\&&'_{\text{pre}}) : (X_{\text{pre}}' Y_1) \Rightarrow (X_{\text{pre}}' Y_2) \Rightarrow (X_{\text{pre}}' \langle Y_1, Y_2 \rangle) & \text{lazy}'_{\text{pre}} : (1 \Rightarrow (X_{\text{pre}}' Y)) \Rightarrow (X_{\text{pre}}' Y) \\
(h_1 \&&'_{\text{pre}} h_2) A := \langle h_1 A, h_2 A \rangle'_{\text{pre}} & \text{lazy}'_{\text{pre}} h A := \text{if } (A = \emptyset) \emptyset'_{\text{pre}} (h 0 A)
\end{array}$$

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

$$\begin{array}{ll}
X_{\text{pre}^*}' Y ::= \text{AStore } (\Omega \times T) (X_{\text{pre}}' Y) & \text{ifte}'^{\psi}_{\text{pre}^*} : (X_{\text{pre}^*}' \text{Bool}) \Rightarrow (X_{\text{pre}^*}' Y) \Rightarrow (X_{\text{pre}^*}' Y) \Rightarrow (X_{\text{pre}^*}' Y) \\
\text{random}'_{\text{pre}^*} : X_{\text{pre}^*}' [0, 1] & \text{ifte}'^{\psi}_{\text{pre}^*} k_1 k_2 k_3 j := \\
\text{random}'_{\text{pre}^*} j := & \text{let } \langle C_k, p_k \rangle := k_1 (\text{left } j) A \\
& \quad \langle C_b, p_b \rangle := \text{branch}_{\text{pre}^*} j A \\
& \quad C_2 := C_k \cap C_b \cap \{\text{true}\} \\
& \quad C_3 := C_k \cap C_b \cap \{\text{false}\} \\
& \quad A_2 := p_k C_2 \cap p_b C_2 \\
& \quad A_3 := p_k C_3 \cap p_b C_3 \\
& \quad \text{in case } C_b \\
& \quad \{\text{true}, \text{false}\} \longrightarrow \langle \top, \lambda B. A_2 \vee A_3 \rangle \\
& \quad \{\text{true}\} \longrightarrow k_2 (\text{left } (\text{right } j)) A_2 \\
& \quad \{\text{false}\} \longrightarrow k_3 (\text{right } (\text{right } j)) A_3
\end{array}$$

(c) Preimage* arrow combinators for probabilistic choice and guaranteed termination. Figure 7.8 (AStore arrow transformer) defines η'_{pre^*} , (\ggg'_{pre^*}) , $(\&&'_{\text{pre}^*})$, $\text{ifte}'_{\text{pre}^*}$ and $\text{lazy}'_{\text{pre}^*}$.

Figure 7.12: Implementable arrows that approximate preimage arrows. Specific lifts such as $\text{fst}'_{\text{pre}} := \text{arr}'_{\text{pre}} \text{fst}$ are computable (see Figure 7.10), but arr'_{pre} is not.

7.9.3 Correctness

From here on, $\llbracket \cdot \rrbracket'_{\text{pre}^*}$ interprets programs as approximating preimage* arrow computations using $\text{ifte}'^{\psi}_{\text{pre}^*}$. The following theorems assume $h := \llbracket e \rrbracket'_{\text{pre}^*} : X_{\text{pre}^*}' Y$ and $h' := \llbracket e \rrbracket'_{\text{pre}^*} : X_{\text{pre}^*}' Y$

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 A.28). Because $\text{ifte}'_{\text{pre}^*}$ does not branch when either branch could be taken, an equivalent finite expression exists for each rectangular domain subset A .

Theorem 7.57 (equivalent finite expression). *For all $A \in \text{Rect } \langle\langle \Omega, T \rangle, X \rangle$, there exists a finite expression e' for which, if $h'' := \llbracket e' \rrbracket_{\text{pre}^*}^{\psi'}$, then $\text{ap}'_{\text{pre}}(h'' j_0 A) B = \text{ap}'_{\text{pre}}(h' j_0 A) B$ for all $B \in \text{Rect } Y$.*

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

Corollary 7.58 (terminating). *For all $A \in \text{Rect } \langle\langle \Omega, T \rangle, X \rangle$ and $B \in \text{Rect } Y$, $\text{ap}'_{\text{pre}}(h' j_0 A) B$ terminates.*

Theorem 7.59 (sound). *For all $A \in \text{Rect } \langle\langle \Omega, T \rangle, X \rangle$ and $B \in \text{Rect } Y$, $\text{ap}_{\text{pre}}(h j_0 A) B \subseteq \text{ap}'_{\text{pre}}(h' j_0 A) B$.*

Proof. By construction and Corollary 7.58 (recall non-“ \equiv ” statements imply termination). \square

Theorem 7.60 (monotone). *$\text{ap}'_{\text{pre}}(h' j_0 A) B$ is monotone in both A and B .*

Proof. Lattice operators (\cap) and (\vee) are monotone, as is (\times) . Therefore, id_{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 η'_{pre^*} ; therefore id_{pre^*} and other lifts are monotone.

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

For monotonicity in A , suppose $A_1 \subseteq A_2$. Apply Theorem 7.57 with A_1 to yield e' ; clearly, it is also an equivalent finite expression for A_2 . Monotonicity follows from structural induction on the interpretation of e' .

For monotonicity in B , apply Theorem 7.57 with a fixed A . \square

Theorem 7.61 (decreasing). *If $A \in \text{Rect } \langle\langle \Omega, T \rangle\rangle, X$ and $B \in \text{Rect } Y$, $\text{ap}'_{\text{pre}} (h' j_0 A) B \subseteq A$.*

Proof. Because they compute exact preimages of rectangular sets under restriction to rectangular domains, id_{pre} and the other lifts in Figure 7.10 are decreasing.

By definition and applying basic lattice properties,

$$\begin{aligned} \text{ap}'_{\text{pre}} ((h_1 \ggg'_{\text{pre}} h_2) A) B &\equiv \text{ap}'_{\text{pre}} (h_1 A) B' \text{ for some } B' \\ \text{ap}'_{\text{pre}} ((h_1 \&&'_\text{pre} h_2) A) B &\equiv \text{ap}'_{\text{pre}} (h_1 A) (\text{proj}_1 B) \cap \text{ap}'_{\text{pre}} (h_2 A) (\text{proj}_2 B) \\ \text{ap}'_{\text{pre}} (\text{ifte}'_{\text{pre}} h_1 h_2 h_3 A) B &\equiv \text{let } A_2 := \text{ap}'_{\text{pre}} (h_1 A) \{\text{true}\} \\ &\quad A_3 := \text{ap}'_{\text{pre}} (h_1 A) \{\text{false}\} \\ &\quad \text{in } \text{ap}'_{\text{pre}} (h_2 A_2) B \vee \text{ap}'_{\text{pre}} (h_3 A_3) B \\ \text{ap}'_{\text{pre}} (\text{lazy}'_{\text{pre}} h A) B &\equiv \text{if } (A = \emptyset) \oslash (\text{ap}'_{\text{pre}} (h 0 A) B) \end{aligned} \tag{7.73}$$

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. \square

7.9.4 Preimage Refinement Algorithm

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

Definition 7.62 (preimage refinement algorithm). Let $B \in \text{Rect } Y$ and

$$\begin{aligned} \text{refine} : \text{Rect } \langle\langle \Omega, T \rangle, X \rangle &\Rightarrow \text{Rect } \langle\langle \Omega, T \rangle, X \rangle \\ \text{refine } A &:= \text{ap}'_{\text{pre}} (h' j_0 A) B \end{aligned} \tag{7.74}$$

Define $\text{split} : \text{Rect } \langle\langle \Omega, T \rangle, X \rangle \Rightarrow \text{Set} (\text{Rect } \langle\langle \Omega, T \rangle, X \rangle)$ to produce positive-measure, disjoint rectangles, and define

$$\begin{aligned} \text{refine}^* : \text{Set} (\text{Rect } \langle\langle \Omega, T \rangle, X \rangle) &\Rightarrow \text{Set} (\text{Rect } \langle\langle \Omega, T \rangle, X \rangle) \\ \text{refine}^* A &:= \text{image refine } (\bigcup_{A \in \mathcal{A}} \text{split } A) \end{aligned} \tag{7.75}$$

For any positive-measure $A_0 \in \text{Rect } \langle\langle \Omega, T \rangle, X \rangle$, iterate refine^* on $\{A_0\}$.

Figure 7.13 illustrates the preimage refinement algorithm.

Theorem 7.61 (decreasing) guarantees $\text{refine } A$ is never larger than A . Theorem 7.60 (monotone) guarantees refining a *partition* of A never does worse than refining A itself. Theorem 7.59 (sound) guarantees the algorithm is sound: the exact preimage of B is always contained in the covering partition 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 `rational? random`, where `rational?` returns `true` when its argument is rational and loops otherwise. (This is definable using a (\leq) primitive.) The preimage of $\{\text{true}\}$ (the rational numbers) has measure 0, but its Jordan outer measure is 1.

We conjecture that a minimal requirement for preimage refinement's measures to converge is that 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.

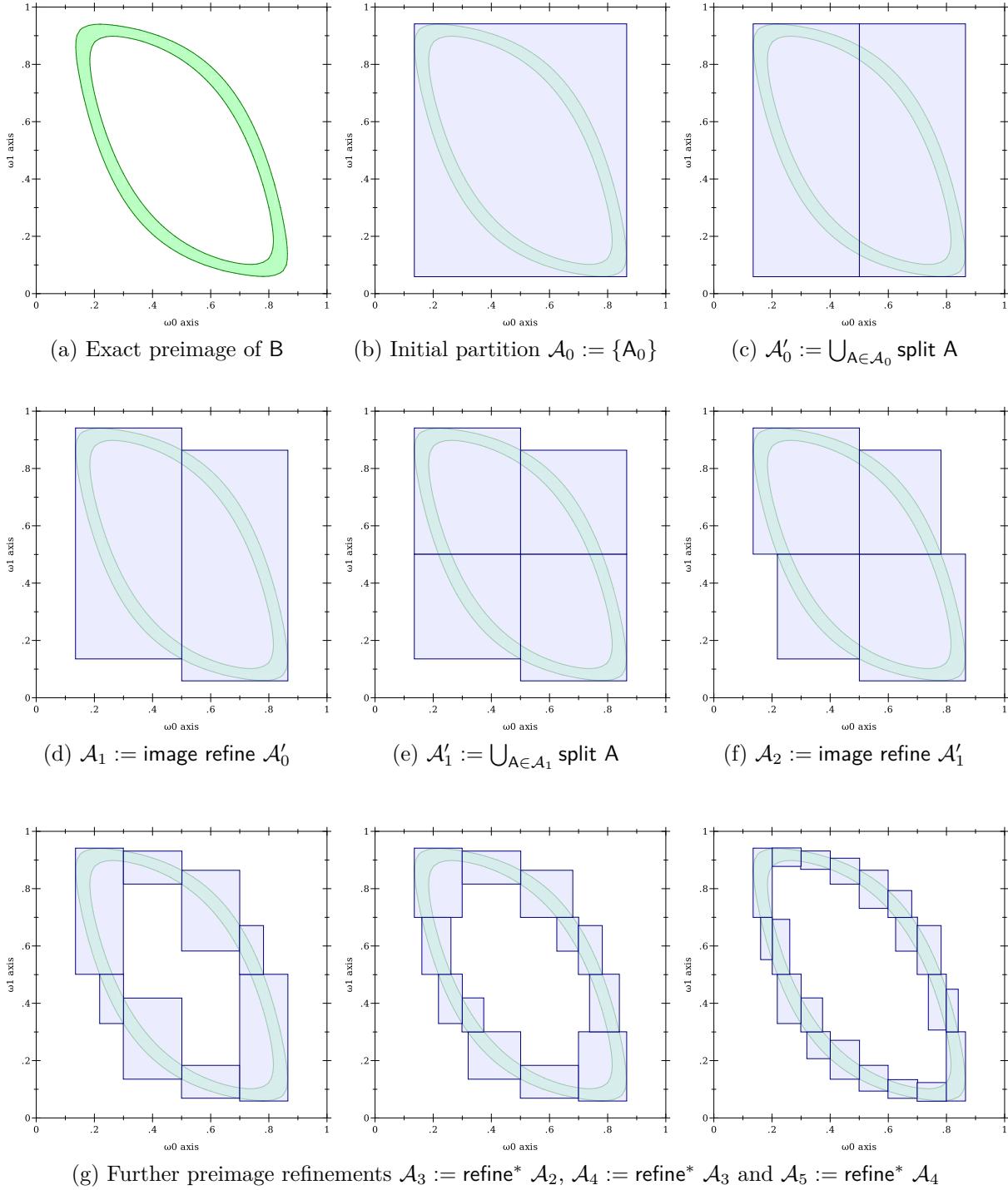


Figure 7.13: Preimage refinement algorithm on $\langle\langle\Omega, T\rangle, X\rangle$. Only two dimensions of Ω are shown. In this example, the covering partition appears to converge in measure to the exact. (In the worst case, 8.12a represents an open set, which in the limit, preimage refinement overapproximates only on the boundary.)

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 λ -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 λ -calculus. We have done so in Typed Racket and Haskell [1]. Both implementations' arrow combinator definitions are almost line-for-line transliterations from the figures. They are at <https://github.com/ntoronto/writing/tree/master/2014esop-code>. [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 λ_{ZFC} , which allows such arrows to carry out uncountably infinite computations.

Further, because the approximating semantics targets a computable λ_{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 preceding chapter leaves out some details; in particular, how to

1. Represent and compute with abstract sets.
2. Compute approximate preimages under real primitives (especially $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ functions).
3. Use preimage refinement to compute conditional probabilities efficiently.

In this chapter, we provide these details.

XXX: diagram with implementation components and dependencies

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 X , we need to define, in the host language,

- A type of rectangles of X with an associated type of members of X .
- Representations of the sets \emptyset and 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

```

class Eq s => Set s where
  type Member s
  empty :: s
  univ :: s
  (/\<>) :: s -> s -> s
  (\\<>) :: s -> s -> s
  singleton :: Member s -> s
  member :: Member s -> s -> Bool

```

Figure 8.1: A Haskell typeclass for rectangular sets.

from the rectangular cover of a preimage set lie within the preimage set.

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 `Rect X` is represented by a type instance of `Set`. For example, the code in Figure 8.2 represents `Rect <X1, X2>` 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 X_1, X_2, \dots, X_n are considered as one universal type $X := \bigcup_{i=1}^n X_i$. The same types, values and operations are necessary; i.e. the type of rectangles of X and of values of X , representations of \emptyset and X , intersection, join, singleton, and membership.

Of course, it is good factorization to have separate representations for each type 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`), Bool (`Bool-Set`), $\langle X, X \rangle$ (`Pair-Set`), $\{\langle \rangle\}$ (`Null-Set`), $J \rightarrow [0, 1]$ (`Omega-Set`) and $J \rightarrow \text{Bool}_\perp$ (`Trace-Set`). The type `Nonempty-Set` is the union of these types and `Univ-Set`, which represents X . The `Set` type additionally represents \emptyset . The `Value` type represents members of X and is defined similarly, but mostly uses built-in Racket types such as `Real` and `Pair`.

¹It 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 $\text{Rect}(\langle X_1, X_2 \rangle)$.

The `intersect` function receives any two `Set` instances and dispatches to a more 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 $(A_1 \times A_2) \cap (B_1 \times B_2) = (A_1 \cap B_1) \times (A_2 \cap B_2)$.

Subsets of `Bool` are easy to represent. Subsets of $\{\langle \rangle\}$ are trivial.

```

;; 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.

We need the representation of real sets to have closed intervals because $\Omega = \mathbb{J} \rightarrow [0, 1]$. Because preimage refinement splits Ω 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 \mathbb{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 `f1` function converts an exact rational to a floating-point number by rounding its argument to the nearest one. The logic after `(define b (f1 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 π , 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 $J \rightarrow A$ have only finitely many projections that are proper subsets of A . For example, for $\Omega := J \rightarrow [0, 1]$, if $\Omega' \in \text{Rect } \Omega$, then $\text{proj } j \Omega' \subset [0, 1]$ for only finitely many $j \in J$. Further, the index set 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-similarity to represent Ω rectangles by a finite data structure: a subtree in which every projection is $[0, 1]$ is represented by (the representation of) Ω itself.

We need a constructor for building binary trees recursively. The following function receives a node value a and two tree encodings l and r , and returns a tree encoding that maps j_0 to a , and has l and r as the left and right subtrees.

$$\begin{aligned} \text{tree-node} : A &\Rightarrow (J \rightarrow A) \Rightarrow (J \rightarrow A) \Rightarrow (J \rightarrow A) \\ \text{tree-node } a \mid l \mid r &:= \{ \langle j_0, a \rangle \} \cup \\ &\quad \{ \langle \text{left } j, a \rangle \mid \langle j, a \rangle \in l \} \cup \\ &\quad \{ \langle \text{right } j, a \rangle \mid \langle j, a \rangle \in r \} \end{aligned} \tag{8.1}$$

From `tree-node`, we define a function to construct instances of $\text{Rect } (J \rightarrow A)$ from a projection,

and left and right subtree rectangles. It is essentially a trinary cartesian product.

$$\begin{aligned} \text{tree-prod} : \text{Rect } A &\Rightarrow \text{Rect } (J \rightarrow A) \Rightarrow \text{Rect } (J \rightarrow A) \Rightarrow \text{Rect } (J \rightarrow A) \\ \text{tree-prod } A \ L \ R &:= \{\text{tree-cons } a \mid r \mid a \in A, l \in L, r \in R\} \end{aligned} \tag{8.2}$$

Any $\text{Rect } (J \rightarrow A)$ can be constructed from $J \rightarrow A$ itself, finitely many projections, and finitely many applications of `tree-prod`. For example,

$$\text{tree-prod } [0, \frac{1}{2}] (\text{tree-prod } [\frac{1}{2}, 1] \Omega \Omega) \Omega \tag{8.3}$$

constructs an instance $\Omega' \in \text{Rect } \Omega$ for which $\text{proj } j_0 \Omega' = [0, \frac{1}{2}]$ and $\text{proj } (\text{left } j_0) \Omega' = [\frac{1}{2}, 1]$, and all other projections are $[0, 1]$.

In Figure 8.5, `tree-prod` is represented by a data type `Omega-Node`, and Ω is represented by the singleton value `univ-omega-set`. Representations of $\text{Rect } \Omega$ 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 Ω 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)
```

```

;; 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 Ω , 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.

Figure 8.6 lists an implementation of values in Ω , 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 `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 $\llbracket \cdot \rrbracket_{a^*}^\Downarrow$ by this rule:

$$\llbracket \text{strict-if } e_1 \ e_2 \ e_3 \rrbracket_{a^*}^\Downarrow := \text{ifte}_{a^*} \ \llbracket e_1 \rrbracket_{a^*}^\Downarrow \ \llbracket e_2 \rrbracket_{a^*}^\Downarrow \ \llbracket e_3 \rrbracket_{a^*}^\Downarrow \quad (8.4)$$

Recall that `if` is interpreted using `iftea*` and `lazya*`, 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`:

$$\begin{aligned} & \text{if } e \langle \rangle \text{ random} \\ & \text{strict-if } e \langle \rangle \text{ random} \end{aligned} \quad (8.5)$$

The `if` expression is interpreted as an application of `iftepre*^\Downarrow`, 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 `iftepre*`, 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:

$$\begin{aligned} a \text{ and } b & : \equiv \text{if } a \ b \ \text{false} \\ a \text{ and* } b & : \equiv \text{strict-if } a \ b \ \text{false} \end{aligned} \quad (8.6)$$

Here, “ =: ” denotes defining special syntax rather than defining a function. (Otherwise, both conjunctions would be strict.)

Another is to assert that $\text{prop? } x$ for some predicate prop? and value x :

$$\text{assert prop? } x \text{ :}\equiv \text{strict-if } (\text{prop? } x) \times \text{fail} \quad (8.7)$$

Here, fail is interpreted as a computation that always returns \emptyset for images and preimages. This expression thus restricts the program domain to the set of values for which $\text{prop? } x$ evaluates to true regardless of branch traces, which cannot be done using if .

Another is pasting together piecewise monotone functions (Section 8.3.4).

With strict-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 real? that returns true when its argument is a real number and false otherwise. As a preimage computation, it could be defined as

$$\begin{aligned} \text{real?}_{\text{pre}} A := \text{case } & \langle A \cap \mathbb{R}, A \setminus \mathbb{R} \rangle \\ & \langle \emptyset, \emptyset \rangle \longrightarrow \langle \emptyset, \lambda B. \emptyset \rangle \\ & \langle A_t, \emptyset \rangle \longrightarrow \text{const}_{\text{pre}} \text{ true } A_t \\ & \langle \emptyset, A_f \rangle \longrightarrow \text{const}_{\text{pre}} \text{ false } A_f \\ & \langle A_t, A_f \rangle \longrightarrow \langle \text{Bool}, \lambda B. (\text{if } (\text{true} \in B) A_t \emptyset \cup (\text{if } (\text{false} \in B) A_f \emptyset) \rangle \end{aligned} \quad (8.8)$$

We potentially have a problem implementing this: we do not have relative complement for implementing $A \setminus \mathbb{R}$. If we have a limited number of data types, however, we can do this:

$$A \setminus \mathbb{R} = A \cap (X_1 \cup X_2 \cup \dots \cup X_n) \quad (8.9)$$

where \mathbb{R} does not appear in the union $X_1 \cup X_2 \cup \dots \cup X_n$, 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 $X_1 \cup X_2 \cup \dots \cup X_n$ 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 **real?**_{pre} to arbitrary predicates:

```
predicatepre Xt Xf A :=  

  case ⟨A ∩ Xt, A ∩ Xf⟩  

    ⟨∅, ∅⟩ → ⟨∅, λB. ∅⟩  

    ⟨At, ∅⟩ → constpre true At  

    ⟨∅, Af⟩ → constpre false Af  

    ⟨At, Af⟩ → ⟨Bool, λB. (if (true ∈ B) At ∅) ∪ (if (false ∈ B) Af ∅)⟩
```

(8.10)

Thus, **real?**_{pre} ≡ **predicate**_{pre} \mathbb{R} ($\top \setminus \mathbb{R}$). In the implementation, $\top \setminus \mathbb{R}$ would be represented by an instance of **Top-Union-Set**.

8.2.3 Testing

Dr. Bayes's rectangular sets include sets of booleans, {⟨⟩}, 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) , \emptyset and \top , the algebraic laws are

$$\begin{aligned}
 \emptyset \vee A &= A && (\vee) \text{ identity} \\
 \top \cap A &= A && (\cap) \text{ identity} \\
 A \vee B &= B \vee A && (\vee) \text{ commutativity} \\
 A \cap B &= B \cap A && (\cap) \text{ commutativity} \\
 (A \vee B) \vee C &= A \vee (B \vee C) && (\vee) \text{ associativity} \\
 (A \cap B) \cap C &= A \cap (B \cap C) && (\cap) \text{ associativity} \\
 A \vee (A \cap B) &= A && (\vee)-(\cap) \text{ absorption} \\
 A \cap (A \vee B) &= A && (\cap)-(\vee) \text{ absorption}
 \end{aligned} \tag{8.11}$$

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:

$$\begin{aligned}
 x \in A \text{ or } x \in B &\implies x \in (A \vee B) && (\vee) \text{ membership} \\
 x \in A \text{ and } x \in B &\iff x \in (A \cap B) && (\cap) \text{ membership}
 \end{aligned} \tag{8.12}$$

These are taken from the definitions of (\cup) and (\cap) , but the first has (\iff) replaced by (\implies) because (\vee) overapproximates (i.e. if $x \in (A \vee B)$, it may be in neither A nor B).

If the preceding 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 $A = \emptyset$ be decidable. (Hopefully it is also efficient.) So to decide equality nonstructurally, we implement (\subseteq) as `subseteq?` and use Lemma 4.1:

$$A = B \iff A \subseteq B \text{ and } B \subseteq A \quad (=) \text{ extensionality} \quad (8.13)$$

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:

$$A \subseteq B \iff x \in A \implies x \in B \quad (\subseteq) \text{ definition} \quad (8.14)$$

If the preceding 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 A , B and C .
2. Randomly generate $x \in A$ and $y \in B$.
3. Evaluate the preceding 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 $x \in \top$, then $x \in A$ would be rare, and $x \in A \implies x \in B$ would too often be equivalent to `false` $\implies x \in B$, which is always `true`. Of course, we cannot

always test with $x \in A = \text{true}$, so for more complete coverage we also generate $y \in B$ and test $y \in A \implies y \in B$. 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:

$$\begin{aligned} A \subseteq A && (\subseteq) \text{ reflexivity} \\ A \subseteq B \text{ and } B \subseteq A \implies A = B && (\subseteq) \text{ antisymmetry} \\ A \subseteq B \text{ and } B \subseteq C \implies A \subseteq C && (\subseteq) \text{ transitivity} \end{aligned} \tag{8.15}$$

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:

$$\begin{aligned} B = A \vee B \iff A \subseteq B && (\vee)-(\subseteq) \text{ definition} \\ A = A \cap B \iff A \subseteq B && (\cap)-(\subseteq) \text{ definition} \\ A \subseteq A \vee B && (\vee) \text{ increasing} \\ A \cap B \subseteq A && (\cap) \text{ decreasing} \\ A_1 \subseteq A_2 \text{ and } B_1 \subseteq B_2 \implies A_1 \vee B_1 \subseteq A_2 \vee B_2 && (\vee) \text{ monotone} \\ A_1 \subseteq A_2 \text{ and } B_1 \subseteq B_2 \implies A_1 \cap B_1 \subseteq A_2 \cap B_2 && (\cap) \text{ monotone} \end{aligned} \tag{8.16}$$

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 verified the preceding 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). $\llbracket a_1, a_2, \alpha_1, \alpha_2 \rrbracket$ denotes an interval, where $a_1, a_2 \in \overline{\mathbb{R}}$ are extended real endpoints, and $\alpha_1, \alpha_2 \in \text{Bool}$ determine whether a_1 and a_2 are contained in the interval.

Some intervals, using $\llbracket \cdot, \cdot, \cdot, \cdot \rrbracket$ notation:

$$\begin{aligned} \llbracket 0, 1, \text{true}, \text{false} \rrbracket &= [0, 1) \\ \llbracket -\infty, 0, \text{false}, \text{true} \rrbracket &= (-\infty, 0] \\ \llbracket -\infty, +\infty, \text{false}, \text{false} \rrbracket &= (-\infty, +\infty) = \mathbb{R} \\ \llbracket -\infty, +\infty, \text{true}, \text{true} \rrbracket &= [-\infty, +\infty] = \overline{\mathbb{R}} \end{aligned} \tag{8.17}$$

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).

Lemma 8.2 (strictly monotone, surjective implies invertible, continuous). *If $g : A \rightarrow B$ is strictly monotone, g is injective. If g is additionally surjective, 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 $A' \subseteq A$, $B' \subseteq B$, and $g : A \rightarrow B$ has inverse $g^{-1} : B \rightarrow A$, then preimage (restrict $f|A'$) $B' = A' \cap \text{image } g^{-1}|B'$.*

These facts suggest that we can compute images (or preimages) of intervals under any strictly monotone, surjective g by applying g (or its inverse) to interval endpoints to yield an interval, as in Figure 8.7. This is evident for endpoints in A . Limit endpoints like $+\infty$ require a larger \bar{g} defined on a compact superset of A .

Theorem 8.4 (images of intervals by endpoints). *Let \bar{A} and \bar{B} be compact subsets of $\overline{\mathbb{R}}$, $\bar{g} : \bar{A} \rightarrow \bar{B}$ be strictly monotone and surjective, and g be the restriction of \bar{g} to some $A \subseteq \bar{A}$. For all nonempty $[a_1, a_2, \alpha_1, \alpha_2] \subseteq A$,*

- *If \bar{g} is increasing, $\text{image } g [a_1, a_2, \alpha_1, \alpha_2] = [\bar{g} a_1, \bar{g} a_2, \alpha_1, \alpha_2]$.*
- *If \bar{g} is decreasing, $\text{image } g [a_1, a_2, \alpha_1, \alpha_2] = [\bar{g} a_2, \bar{g} a_1, \alpha_2, \alpha_1]$.*

Proof. Because \bar{A} is compact and totally ordered, every subset of \bar{A} has a lower and an upper bound in \bar{A} . Therefore, the endpoints of every interval subset of A are in \bar{A} .

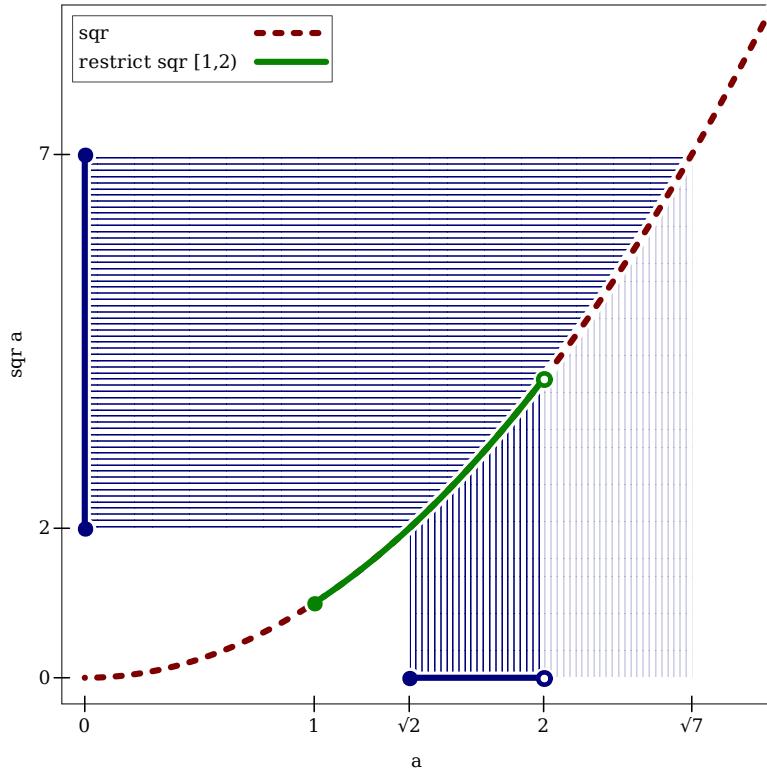


Figure 8.7: Computing the preimage of the interval $[2, 7]$ under sqr restricted to $[1, 2]$, by computing roots and intersecting with $[1, 2]$.

Let $(a_1, a_2] \subseteq A$. Suppose \bar{g} is strictly increasing; thus $a_1 < a \leq a_2$ if and only if $\bar{g} a_1 < \bar{g} a \leq \bar{g} a_2$, so $\text{image } g (a_1, a_2] = \text{image } \bar{g} (a_1, a_2] = (\bar{g} a_1, \bar{g} a_2]$. The remaining cases are similar. \square

To use Theorem 8.4 to compute preimages under g by computing images under its inverse g^{-1} , we must know if g^{-1} is increasing or decreasing. The following lemma can help.

Lemma 8.5 (inverse direction). *If $g : A \rightarrow B$ is strictly monotone and surjective with inverse $g^{-1} : B \rightarrow A$, then g is increasing if and only if g^{-1} is increasing.*

Example 8.6 (nonnegative square). The extension of $\text{sqr}^+ : [0, +\infty) \rightarrow [0, +\infty)$, where $\text{sqr}^+ a := a \cdot a$, to the compact superdomain $[0, +\infty]$ is

$$\begin{aligned} \overline{\text{sqr}^+} &: [0, +\infty] \rightarrow [0, +\infty] \\ \overline{\text{sqr}^+} a &:= \lim_{a' \rightarrow a} \text{sqr}^+ a' = \text{if } (a = +\infty) +\infty (\text{sqr}^+ a) \end{aligned} \tag{8.18}$$

(With respect to \mathbb{R} 's standard topology, which is first-countable, sqr^+ is continuous and thus limit-preserving.) The extension of its inverse sqrt^+ is $\overline{\text{sqrt}^+} : [0, +\infty] \rightarrow [0, +\infty]$, defined similarly, which by Lemma 8.5 is also strictly increasing. Thus,

$$\begin{aligned}
 \text{image } \text{sqr}^+ [5, +\infty) &= [\overline{\text{sqr}^+} 5, \overline{\text{sqr}^+} +\infty) \\
 &= [25, +\infty) \\
 \text{preimage (restrict } \text{sqr}^+ [1, 2)) [2, 7] &= [1, 2) \cap \text{image } \text{sqrt}^+ [2, 7] \tag{8.19} \\
 &= [1, 2) \cap [\overline{\text{sqrt}^+} 2, \overline{\text{sqrt}^+} 7] \\
 &= [1, 2) \cap [\sqrt{2}, \sqrt{7}] \\
 &= [\sqrt{2}, 2)
 \end{aligned}$$

by Theorem 8.4 and Lemma 8.3. \diamond

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 $g_c : A \times B \rightarrow C$. Functions $g_a : B \times C \rightarrow A$ and $g_b : C \times A \rightarrow B$ defined so that*

$$g_c \langle a, b \rangle = c \iff g_a \langle b, c \rangle = a \iff g_b \langle c, a \rangle = b \tag{8.20}$$

*are **axial inverses** with respect to g_c 's first and second arguments.*

We call g_c **axis-invertible** or **trjective** when it has axial inverses g_a and g_b . We call g_a the **first axial inverse** of g_c because it is the inverse of g_c along the first axis: g_a with only c varying, or $\lambda c \in C. g_a \langle b, c \rangle$, is the inverse of g_c with only a varying, or $\lambda a \in A. g_c \langle a, b \rangle$. Similarly, g_b is the **second axial inverse**.

Example 8.8. Let $\text{add}_c : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, $\text{add}_c(a, b) := a + b$. Its axial inverses are $\text{add}_a(b, c) := c - b$ and $\text{add}_b(c, a) := c - a$. \diamond

We have chosen the axial inverse function types carefully: they are the only types for which g_c , g_a and g_b form a cyclic group.

Theorem 8.9 (axial inverse cyclic group). *The following statements are equivalent.*

- g_c has axial inverses g_a and g_b .
- g_a has axial inverses g_b and g_c .
- g_b has axial inverses g_c and g_a .

Equivalently, every axis-invertible function generates a cyclic group of order 3 by inversion in the first axis.

Proof. This is evident from the definition of axial inverse (Definition 8.7). \square

This fact is analogous to how mutual inverses g and g^{-1} form a cyclic group of order 2 generated by inversion. Similar to using mutual inversion to compute images and preimages under both sqr^+ and sqrt^+ , Theorem 8.9 allows computing preimages under two-argument functions related by axial inversion.

Example 8.10. Define $\text{sub}_c : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ by $\text{sub}_c(a, b) := a - b$. Because $\text{sub}_c = \text{add}_b$, $\text{sub}_a = \text{add}_c$ and $\text{sub}_b = \text{add}_a$. \diamond

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 $A' \subseteq A$, $B' \subseteq B$, $C' \subseteq C$, and $g_c : A \times B \rightarrow C$ with axial inverses g_a and g_b . If $g'_c := \text{restrict } g_c(A' \times B')$, then*

$$\text{preimage } g'_c(C') \subseteq (A' \cap \text{image } g_a(B' \times C')) \times (B' \cap \text{image } g_b(C' \times A')) \quad (8.21)$$

Further, the right-hand side is the smallest rectangular superset.

Proof. The smallest rectangle containing preimage $g'_c C'$ is

$$\begin{aligned} \text{preimage } g'_c C' &\subseteq (\text{image } \text{fst } (\text{preimage } g'_c C')) \times \\ &\quad (\text{image } \text{snd } (\text{preimage } g'_c C')) \end{aligned} \tag{8.22}$$

Starting with the first set in the product, expand definitions, distribute fst , replace $g_c \langle a, b \rangle = c$ by $g_a \langle b, c \rangle = a$, and simplify:

$$\begin{aligned} &\text{image } \text{fst } (\text{preimage } g'_c C') \\ &= \text{image } \text{fst } \{ \langle a, b \rangle \in A' \times B' \mid g_c \langle a, b \rangle \in C' \} \\ &= \{ a \in A' \mid \exists b \in B'. g_c \langle a, b \rangle \in C' \} \\ &= \{ a \in A' \mid \exists b \in B', c \in C'. g_c \langle a, b \rangle = c \} \\ &= \{ a \in A' \mid \exists b \in B', c \in C'. g_a \langle b, c \rangle = a \} \\ &= \{ g_a \langle b, c \rangle \mid b \in B', c \in C', g_a \langle b, c \rangle \in A' \} \\ &= A' \cap \{ g_a \langle b, c \rangle \mid b \in B', c \in C' \} \\ &= A' \cap \text{image } g_a (B' \times C') \end{aligned}$$

The second set in the product is similar. \square

Example 8.12. Let $\text{add}'_c := \text{restrict } \text{add}_c ([0, 1] \times [0, 2])$. By Theorem 8.11,

$$\begin{aligned} \text{preimage } \text{add}'_c [0, \frac{1}{2}] &\subseteq ([0, 1] \cap \text{image } \text{add}_a ([0, 2] \times [0, \frac{1}{2}])) \times \\ &\quad ([0, 2] \cap \text{image } \text{add}_b ([0, \frac{1}{2}] \times [0, 1])) \\ &= ([0, 1] \cap [-2, \frac{1}{2}]) \times ([0, 2] \cap [-1, \frac{1}{2}]) \\ &= [0, \frac{1}{2}] \times [0, \frac{1}{2}] \end{aligned}$$

is the smallest rectangular subset of $[0, 1] \times [0, 2]$ containing the preimage of $[0, \frac{1}{2}]$ under add'_c .

Figure 8.8 illustrates the calculation. \diamond

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

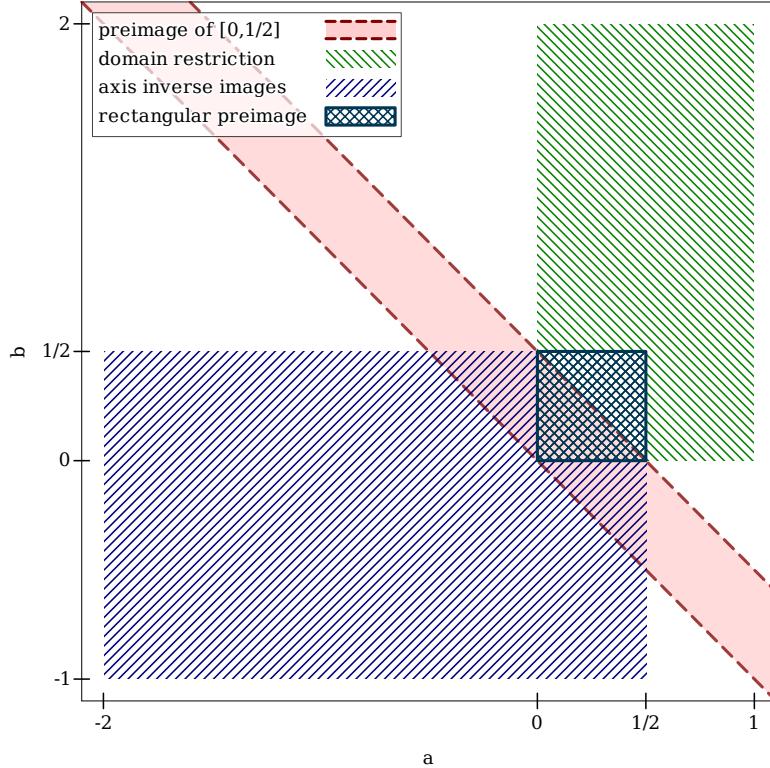


Figure 8.8: Computing an approximate preimage of $[0, \frac{1}{2}]$ under addition restricted to $[0, 1] \times [0, 2]$ (Example 8.12). The preimage is approximated by intersecting the domain with an overapproximation computed using axial inverses.

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).

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 $g_c : A \times B \rightarrow C$ has property P **in its first axis** when $P(\lambda a \in A. g_c(a, b))$ for all $b \in B$. Similarly, g_c has property P **in its second axis** when $P(\lambda b \in B. g_c(a, b))$ for all $a \in A$.

Now Lemma 8.2's analogue is an easy corollary.

Theorem 8.13 (strictly monotone, surjective implies axis-invertible, continuous). *Let $g_c : A \times B \rightarrow C$ for totally ordered A , B and C . If g_c is surjective and either strictly increasing or strictly decreasing in each axis, then g_c is axis-invertible; further, it and its axial inverses are continuous.*

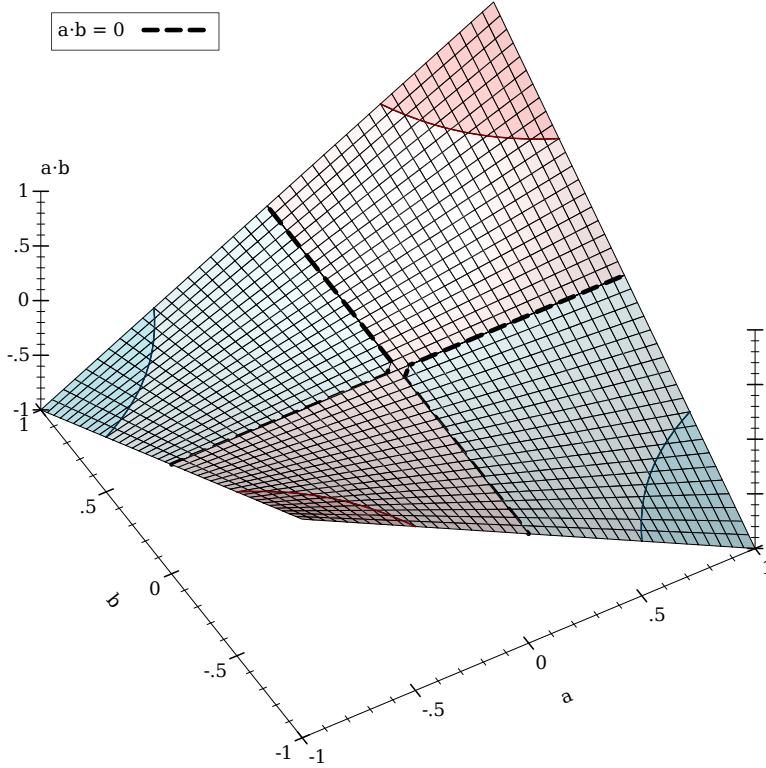


Figure 8.9: Multiplication on $\mathbb{R} \times \mathbb{R}$ is not surjective nor strictly monotone in each axis: $a \cdot 0 = 0$ and $0 \cdot b = 0$ for all a and b (Example 8.16). Fortunately, restricted to each open quadrant, multiplication is surjective and strictly increasing or decreasing in each axis.

Proof. XXX: todo □

Example 8.14. In each axis, add_c is surjective and strictly increasing. In each axis, sub_c is surjective, and is strictly increasing/decreasing in its first/second axis. Therefore, both are axis-invertible. ◇

Restriction usually makes a function not surjective in each axis.

Example 8.15. Let $\text{add}'_c : [0, 1] \times [0, 1] \rightarrow [0, 2]$, defined by restricting add_c . It is surjective, but not in each axis: the range of $\lambda b \in B. \text{add}'_c \langle 0, b \rangle$ is $[0, 1]$, not $[0, 2]$. ◇

Fortunately, restriction sometimes does the opposite.

Example 8.16. Define $\text{mul}_c : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ by $\text{mul}_c \langle a, b \rangle := a \cdot b$. It is not surjective nor strictly monotone in each axis because $\text{mul}_c \langle 0, b \rangle = 0$ for all $b \in B$. (See Figure 8.9.) But

$\text{mul}_c^{++} : (0, +\infty) \times (0, +\infty) \rightarrow (0, +\infty)$, and mul_c restricted to the other open quadrants, are surjective and strictly increasing or decreasing in each axis. \diamond

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. add_c cannot be extended to $\overline{\text{add}}_c : \overline{\mathbb{R}} \times \overline{\mathbb{R}} \rightarrow \overline{\mathbb{R}}$ in the same way sqr^+ is extended to $\overline{\text{sqr}^+}$ because

$$\lim_{(a', b') \rightarrow (a, b)} \text{add}_c \langle a', b' \rangle \quad (8.23)$$

diverges when $\langle a, b \rangle$ is $\langle -\infty, +\infty \rangle$ or $\langle +\infty, -\infty \rangle$. \diamond

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 A, B, C be open subsets of \mathbb{R} , and $g_c : A \times B \rightarrow C$ be surjective and strictly increasing or decreasing in each axis. Let $\overline{A}, \overline{B}$ and \overline{C} be the closures of A, B and C in $\overline{\mathbb{R}}$. The following extension is well-defined:*

$$\begin{aligned} \overline{g_c} : (\overline{A} \times \overline{B}) \setminus N &\rightarrow \overline{C} \\ \overline{g_c} \langle a, b \rangle &:= \lim_{(a', b') \rightarrow (a, b)} g_c \langle a', b' \rangle \end{aligned} \quad (8.24)$$

where $N := \{(\min \overline{A}, \max \overline{B}), (\max \overline{A}, \min \overline{B})\}$ if g_c is increasing in each axis or decreasing in each axis, and $N := \{(\min \overline{A}, \min \overline{B}), (\max \overline{A}, \max \overline{B})\}$ if g_c is increasing/decreasing or decreasing/increasing.

Proof. Suppose g_c is increasing/increasing, and let $\mathbf{xs} : \mathbb{N} \rightarrow A \times B$ be a sequence of g_c 's domain values, and $\mathbf{ys} := \text{map } g_c \mathbf{xs}$.

Interior case: \mathbf{xs} converges to $\langle a, b \rangle \in A \times B$. The limit of \mathbf{ys} is $g_c \langle a, b \rangle$ because g_c preserves limits by its continuity (by Theorem 8.13) in the first-countable space $\mathbb{R} \times \mathbb{R}$.

Corner case: xs converges to $\langle \max \bar{A}, \max \bar{B} \rangle$. It thus has a strictly increasing subsequence. By monotonicity, ys has a strictly increasing subsequence. Because ys is bounded by $\max \bar{C}$, $\overline{g_c} \langle \max \bar{A}, \max \bar{B} \rangle = \max \bar{C}$. A similar argument proves $\overline{g_c} \langle \min \bar{A}, \min \bar{B} \rangle = \min \bar{C}$.

Border case: xs converges to $\langle \max \bar{A}, b' \rangle$ for some $b' \in B$. Define

$$\text{xs}' := \text{map } (\lambda \langle a, b \rangle. \langle g_a \langle b', g_c \langle a, b \rangle \rangle, b' \rangle) \text{ xs} \quad (8.25)$$

where g_a is g_c 's first axial inverse. Now $\text{ys} = \text{map } g_c \text{ xs}'$. Because xs' has a subsequence that is strictly increasing in the first of each pair, and because the second of each pair is the constant b' , by monotonicity, ys has a strictly increasing subsequence. It is bounded by $\max \bar{C}$, so $\overline{g_c} \langle \max \bar{A}, b' \rangle = \max \bar{C}$. Similar arguments prove $\overline{g_c} \langle \min \bar{A}, b' \rangle = \min \bar{C}$, $\overline{g_c} \langle a', \max \bar{B} \rangle = \max \bar{C}$, and $\overline{g_c} \langle a', \min \bar{B} \rangle = \min \bar{C}$.

The cases for g_c 's other possible directions are similar. \square

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 $\text{pow}_c : (0, 1) \times (0, +\infty) \rightarrow (0, 1)$ by $\text{pow}_c \langle a, b \rangle := \exp(b \cdot \log a)$, which is increasing/decreasing. Its extension to a subset of $\bar{\mathbb{R}} \times \bar{\mathbb{R}}$ is

$$\begin{aligned} \overline{\text{pow}_c} : ([0, 1] \times [0, +\infty]) \setminus N &\rightarrow [0, 1] \\ \overline{\text{pow}_c} \langle a, b \rangle &:= \text{case } \langle a, b \rangle \\ &\quad \begin{array}{ll} \langle 0, +\infty \rangle &\longrightarrow 0 \\ \langle 1, 0 \rangle &\longrightarrow 1 \\ \langle 0, b \rangle &\longrightarrow 0 \\ \langle 1, b \rangle &\longrightarrow 1 \\ \langle a, 0 \rangle &\longrightarrow 1 \\ \langle a, +\infty \rangle &\longrightarrow 0 \\ \text{else} &\longrightarrow \text{pow}_c \langle a, b \rangle \end{array} \end{aligned} \quad (8.26)$$

where $N := \{\langle 0, 0 \rangle, \langle 1, +\infty \rangle\}$. \diamond

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 $\text{inc}_1 : (A \times B \rightarrow C) \Rightarrow$

`Bool` so that $\text{inc}_1 g$ if and only if g is strictly increasing in its first axis, and similarly inc_2 so that $\text{inc}_2 g$ if and only if g is strictly increasing in its second axis.

Theorem 8.20 (images of rectangles by interval endpoints). *Let A, B, C be open subsets of \mathbb{R} , and $g_c : A \times B \rightarrow C$ be surjective and strictly increasing or decreasing in each axis, with \overline{g}_c as defined in Theorem 8.18. If $A' := [a_1, a_2, \alpha_1, \alpha_2] \subseteq A$ and $B' := [b_1, b_2, \beta_1, \beta_2] \subseteq B$, then*

$$\begin{aligned} C' &:= \text{image } g_c ([a_1, a_2, \alpha_1, \alpha_2] \times [b_1, b_2, \beta_1, \beta_2]) \\ &= \text{let } \langle a'_1, a'_2, \alpha'_1, \alpha'_2 \rangle := \text{cond } (\text{inc}_1 g_c) \longrightarrow \langle a_1, a_2, \alpha_1, \alpha_2 \rangle \\ &\quad \text{else } \longrightarrow \langle a_2, a_1, \alpha_2, \alpha_1 \rangle \quad (8.27) \\ &\quad \langle b'_1, b'_2, \beta'_1, \beta'_2 \rangle := \text{cond } (\text{inc}_2 g_c) \longrightarrow \langle b_1, b_2, \beta_1, \beta_2 \rangle \\ &\quad \text{else } \longrightarrow \langle b_2, b_1, \beta_2, \beta_1 \rangle \\ &\text{in } [\overline{g}_c \langle a'_1, b'_1 \rangle, \overline{g}_c \langle a'_2, b'_2 \rangle, \alpha'_1 \text{ and } \beta'_1, \alpha'_2 \text{ and } \beta'_2] \end{aligned}$$

Proof. Because g_c is continuous and $A' \times B'$ is a connected set, 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 g_c is increasing/increasing. In this case, $a'_1 = a_1$, $b'_1 = b_1$, and so on. By monotonicity, C' is contained in $[\overline{g}_c \langle a'_1, b'_1 \rangle, \overline{g}_c \langle a'_2, b'_2 \rangle]$. If α'_1 or β'_1 is `false`, C' cannot contain $\overline{g}_c \langle a'_1, b'_1 \rangle$. If α'_2 or β'_2 is `false`, C' cannot contain $\overline{g}_c \langle a'_2, b'_2 \rangle$. Therefore $C' = [\overline{g}_c \langle a'_1, b'_1 \rangle, \overline{g}_c \langle a'_2, b'_2 \rangle, \alpha'_1 \text{ and } \beta'_1, \alpha'_2 \text{ and } \beta'_2]$.

We still must prove $\langle a'_1, b'_1 \rangle$ and $\langle a'_2, b'_2 \rangle$ are in \overline{g}_c 's domain. First, recall $\overline{g}_c : (\overline{A} \times \overline{B}) \setminus N \rightarrow \overline{C}$, where \overline{A} , \overline{B} and \overline{C} are the closures of A , B and C , and $N = \{\langle \min \overline{A}, \max \overline{B} \rangle, \langle \max \overline{A}, \min \overline{B} \rangle\}$. Because $A' \subseteq A$ and $B' \subseteq B$, and A and B are open sets, $a_1 \neq \max \overline{A}$, $a_2 \neq \min \overline{A}$, $b_1 \neq \max \overline{B}$, and $b_2 \neq \min \overline{B}$, so for all $a \in \overline{A}$ and $b \in \overline{B}$,

$$\begin{aligned} \langle a_1, b_1 \rangle &\neq \langle \max \overline{A}, b \rangle & \langle a_2, b_2 \rangle &\neq \langle \min \overline{A}, b \rangle \\ \langle a_1, b_1 \rangle &\neq \langle a, \max \overline{B} \rangle & \langle a_2, b_2 \rangle &\neq \langle a, \min \overline{B} \rangle \end{aligned} \quad (8.28)$$

Therefore, $\langle a_1, b_1 \rangle \notin N$ and $\langle a_2, b_2 \rangle \notin N$, as desired.

The remaining cases for g_c are similar. \square

Example 8.21. Because $\text{inc}_1 \text{ pow}_c$ and not $(\text{inc}_2 \text{ pow}_c)$,

$$\begin{aligned}
& \text{image } \text{pow}_c ((0, \frac{1}{2}] \times [2, +\infty)) \\
&= \text{let } \langle a_1, a_2, \alpha_1, \alpha_2 \rangle := \langle 0, \frac{1}{2}, \text{false}, \text{true} \rangle \\
&\quad \langle b_1, b_2, \beta_1, \beta_2 \rangle := \langle +\infty, 2, \text{false}, \text{true} \rangle \\
&\quad \text{in } [\overline{\text{pow}_c} \langle a_1, b_1 \rangle, \overline{\text{pow}_c} \langle a_2, b_2 \rangle, \alpha_1 \text{ and } \beta_1, \alpha_2 \text{ and } \beta_2] \\
&= [\overline{\text{pow}_c} \langle 0, +\infty \rangle, \overline{\text{pow}_c} \langle \frac{1}{2}, 2 \rangle, \text{false and false, true and true}] \\
&= [0, \frac{1}{4}, \text{false, true}] \\
&= (0, \frac{1}{4}]
\end{aligned}$$

◇

To use Theorem 8.20 to compute approximate preimages under some g_c by computing images under its axial inverses, we must know whether each axis of g_a and 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 $g_c : A \times B \rightarrow C$ be surjective and strictly increasing or decreasing in each axis, with axial inverses g_a and g_b . Then*

1. $\text{inc}_1 g_a$ if and only if $(\text{inc}_1 g_c) \text{ xor } (\text{inc}_2 g_c)$.
2. $\text{inc}_2 g_a$ if and only if $\text{inc}_1 g_c$.

Proof. For 1, let $c \in C$, $b_1, b_2 \in B$, $a_1 := g_a \langle b_1, c \rangle$ and $a_2 := g_a \langle b_2, c \rangle$. Let $c' := g_c \langle a_1, b_2 \rangle$; note $c = g_c \langle a_1, b_1 \rangle = g_c \langle a_2, b_2 \rangle$. Suppose $\text{inc}_1 g_c$ and $\text{inc}_2 g_c$; then $a_1 > a_2 \iff c < c'$ and $b_1 < b_2 \iff c < c'$, so $b_1 < b_2 \iff a_1 > a_2$. The remaining cases are similar.

For statement 2, fix $b \in B$ and apply Lemma 8.5. □

By Theorem 8.22, we can use g_c 's axis directions to determine g_a 's, and by Theorem 8.9 (axial inverse cyclic group), use g_a 's to determine 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 a , restricts Ω at index j to $[a, a]$ using $\Omega' := \text{unproj } j \ \Omega [a, a]$, and computes a preimage under the program's interpretation as a function, restricted to Ω' . If in the forward pass, the approximation of the image of Ω' 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

$$\begin{aligned} \text{sqrt}_{\text{pre}} : [0, +\infty) &\xrightarrow{\text{pre}} [0, +\infty) \\ \text{sqrt}_{\text{pre}} A &:= \langle \text{image } \text{sqrt}^+ A, \text{preimage } (\text{restrict } \text{sqrt}^+ A) \rangle \end{aligned} \tag{8.29}$$

Suppose $A = [\frac{1}{2}, \frac{1}{2}]$, and that the implementation mistakenly computes $\text{image } \text{sqrt}^+ [\frac{1}{2}, \frac{1}{2}]$ 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 $\text{restrict } \text{sqrt}^+ [\frac{1}{2}, \frac{1}{2}]$. By Lemma 8.3, the implementation of `pre` should compute

$$\begin{aligned} \text{preimage } (\text{restrict } \text{sqrt}^+ [\frac{1}{2}, \frac{1}{2}]) (\mathbb{R} \cap [0.7071067811865476, 0.7071067811865476]) \\ = [\frac{1}{2}, \frac{1}{2}] \cap \text{image } \text{sqrt}^+ [0.7071067811865476, 0.7071067811865476] \end{aligned} \tag{8.30}$$

If it again uses compliant floating-point arithmetic but does not round outward, it computes

$$[\frac{1}{2}, \frac{1}{2}] \cap [0.5000000000000001, 0.5000000000000001] = \emptyset \tag{8.31}$$

In fact, an implementation that does not round intervals outward would falsely compute \emptyset

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, `exp` and `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 F_N^{-1} and (its inverse F_N) to transform uniformly distributed random numbers (i.e. each ω_j) into normally distributed random numbers. As a preimage arrow computation, it is

$$\begin{aligned} \text{normal-inv-cdf}_{\text{pre}} : (0, 1) &\rightsquigarrow \mathbb{R} \\ \text{normal-inv-cdf}_{\text{pre}} A &:= \langle \text{image } F_N^{-1} A, \text{preimage } (\text{restrict } F_N^{-1} A) \rangle \end{aligned} \tag{8.32}$$

Racket's `math/distributions` library implements F_N^{-1} with `flnormal-inv-cdf` and F_N with `flnormal-cdf`, whose outputs are always within four floating-point numbers of the exact outputs. The implementation of `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 `f1` is conventional, so the name of the floating-point addition function that rounds down is `f1+/rndd`, and the name of the floating-point square root function that rounds up is `f1sqrt/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
             flsqrt/rndd flsqrt/rndu
             flsqrt/rnddd flsqrt/rnddu))

(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 $\overline{\text{recip}^+}$ be the extension of $\text{recip}^+ : (0, +\infty) \rightarrow (0, +\infty)$ to the compact superdomain $[0, +\infty]$, defined by

$$\begin{aligned} \overline{\text{recip}^+} &: [0, +\infty] \rightarrow [0, +\infty] \\ \overline{\text{recip}^+} a := \lim_{a' \rightarrow a} \text{recip}^+ a' &= \begin{cases} \text{case } a & \\ 0 & \rightarrow +\infty \\ +\infty & \rightarrow 0 \\ \text{else} & \rightarrow \text{recip}^+ a \end{cases} \end{aligned} \tag{8.33}$$

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/rnnd : (Flonum -> Flonum)] [gb/rndu : (Flonum -> Flonum)]
   [ga/rnnd : (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/rnnd gb/rndu ga/rnnd ga/rndu) g)
  (Bijection inc? Y X ga/rnnd ga/rndu gb/rnnd 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/rnnd g/rndu A)
  (match-define (Real-Set a1 a2 a1? a2?) A)
  (cond [inc? (Real-Set (g/rnnd a1) (g/rndu a2) a1? a2?)]
        [else (Real-Set (g/rnnd 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/rnnd gb/rndu _ _) g)
  (let ([A (real-set-intersect A X)])
    (if (empty-set? A)
        empty-set
        (real-set-intersect Y (real-image inc? gb/rnnd 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/rnnd 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/rnnd 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//rnnd 1.0 a)) (λ (a) (fl//rndu 1.0 a))
              (λ (a) (fl//rnnd 1.0 a)) (λ (a) (fl//rndu 1.0 a))))

```

Because recip^+ is surjective, $\text{image } \text{recip}^+ (0, +\infty) = (0, +\infty)$. 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 `(f1/ 1.0 +0.0)` returns `+inf.0`, but `(f1/ 1.0 -0.0)` returns `-inf.0`, as per the IEEE 754 floating-point standard. The implementation should compute

$$\text{image recip}^+ (0, +\infty) = (\overline{\text{recip}^+} +\infty, \overline{\text{recip}^+} 0) = (0, +\infty) \quad (8.34)$$

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 (f1/ 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
              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 `f1rev-/rndd` implements $\text{add}_a \langle b, c \rangle := c - b$ with rounding down.

8.3.4 Piecewise Monotone Primitives

Using `iftepre`, 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

$$\begin{aligned} \text{negative?}_{\text{pre}} &: \mathbb{R} \xrightarrow{\text{pre}} \text{Bool} \\ \text{negative?}_{\text{pre}} &:= \text{predicate}_{\text{pre}} (-\infty, 0] (0, +\infty) \end{aligned} \tag{8.35}$$

as well as `positive?pre` in the same way.

From `sqrpre+` (nonnegative square) and `negpre` (negation) primitives, we define

$$\begin{aligned} \text{sqr}_{\text{pre}} &:= \text{ifte}_{\text{pre}} \text{ negative?}_{\text{pre}} (\text{neg}_{\text{pre}} \ggg_{\text{pre}} \text{sqr}_{\text{pre}}^+) \text{sqr}_{\text{pre}}^+ \\ \text{sqr}_{\text{pre}^*} &:= \eta_{\text{pre}^*} \text{sqr}_{\text{pre}} \end{aligned} \tag{8.36}$$

We then extend $\llbracket \cdot \rrbracket_{\text{pre}^*}^\Downarrow$ by the rule $\llbracket \text{sqr } e \rrbracket_{\text{pre}^*}^\Downarrow := \llbracket e \rrbracket_{\text{pre}^*}^\Downarrow \ggg_{\text{pre}^*} \text{sqr}_{\text{pre}^*}$. Equivalently, we could provide `sqr+`, `neg` and `negative?` as primitives and define

$$\text{sqr } a := \text{strict-if} (\text{negative? } a) (\text{sqr}^+ (\text{neg } a)) (\text{sqr}^+ a) \tag{8.37}$$

as a standard library function for probabilistic programs.

From implementations of multiplication restricted to each open quadrant, or `mulpre++`, `mulpre+-`, `mulpre-+` and `mulpre--`, we can define multiplication on all of $\mathbb{R} \times \mathbb{R}$ with

$$\begin{aligned} \text{mul}_{\text{pre}} &:= \text{ifte}_{\text{pre}} (\text{fst}_{\text{pre}} \ggg_{\text{pre}} \text{positive?}_{\text{pre}}) \\ &\quad (\text{ifte}_{\text{pre}} (\text{snd}_{\text{pre}} \ggg_{\text{pre}} \text{positive?}_{\text{pre}}) \\ &\quad \quad \text{mul}_{\text{pre}}^{++} \\ &\quad (\text{ifte}_{\text{pre}} (\text{snd}_{\text{pre}} \ggg_{\text{pre}} \text{negative?}_{\text{pre}}) \\ &\quad \quad \text{mul}_{\text{pre}}^{+-} \\ &\quad \quad (\text{const}_{\text{pre}} 0))) \\ &\dots [\text{similar code using } \text{mul}_{\text{pre}}^{-+} \text{ and } \text{mul}_{\text{pre}}^{--}] \dots \end{aligned} \tag{8.38}$$

$$\text{mul}_{\text{pre}^*} := \eta_{\text{pre}^*} \text{mul}_{\text{pre}}$$

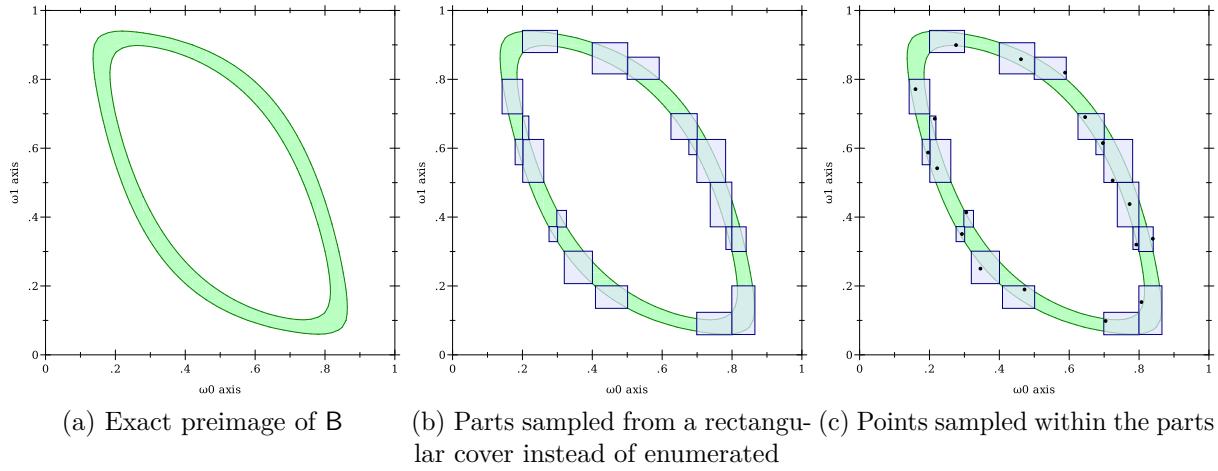


Figure 8.12: A more efficient alternative to the preimage refinement algorithm. Instead of enumerating a rectangular cover, its parts are sampled, and each part is then sampled from. Points that lie outside the exact preimage (which is easy to test) are rejected.

and extend $\llbracket \cdot \rrbracket_{\text{pre}^*}^\downarrow$ by the rule $\llbracket e_1 \cdot e_2 \rrbracket_{\text{pre}^*}^\downarrow := \llbracket \langle e_1, e_2 \rangle \rrbracket_{\text{pre}^*}^\downarrow \ggg_{\text{pre}^*} \text{mul}_{\text{pre}^*}$.

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 Ω . Splitting each axis into only 4 disjoint intervals yields a partition of Ω of size $4^{10} = 1,048,576$.

Fortunately, Bayesian practitioners tend to be satisfied 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 $g := \llbracket p \rrbracket_{\text{map}^*}^\downarrow$, we can answer almost any query $\Pr[B' | B]$ by sampling within $A := \text{preimage } g B$, if the probability of A is positive.

It is easy to sample within A by sampling within Ω and rejecting samples not in A . To

determine whether samples are in A , we can use the interpretation of the program p as a bottom* arrow computation $f := \llbracket p \rrbracket_{\perp^*}^\perp$. 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 A , as computed by preimage refinement, instead of within Ω . 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 X, P by first sampling a part from a partition of X and then sampling from that part.²

First, to restrict probability measures to measurable, positive-probability sets and renormalize them, we define

$$\begin{aligned} \text{condition} : \text{Set } X \rightarrow [0, 1] &\Rightarrow \text{Set } X \Rightarrow \text{Set } X \rightarrow [0, 1] \\ \text{condition } P A &:= \lambda A' \in \text{domain } P. P(A' \cap A) / P A \end{aligned} \tag{8.39}$$

Definition 8.23 (partitioned sampling). *Let X, P be an arbitrary probability space, N be an at-most-countable index set, and $s : N \rightarrow \text{Set } X$ be a partition of X into $|N|$ measurable parts. The following procedure samples from X :*

1. Choose $n \in N$ with probability $P(s n)$.
2. Choose $a \in s n$ according to condition $P(s n)$.

It is not hard to show that partitioned sampling chooses an $a \in X$ according to P .

Example 8.24 (partitioned sampling from a standard normal). Let P be the standard normal distribution's probability measure. To sample according to P , let $N := \{\text{neg}, \text{pos}\}$ and

²This is not *stratified* sampling, which samples a fixed number of times from each partition.

$s = [\text{neg} \mapsto (-\infty, 0], \text{pos} \mapsto (0, +\infty)]$, and define $Q : N \rightarrow \text{Set } \mathbb{R} \rightharpoonup [0, 1]$ by

$$\begin{aligned} Q \text{ neg } A &= P((-\infty, 0] \cap A) / \frac{1}{2} \\ Q \text{ pos } A &= P((0, +\infty) \cap A) / \frac{1}{2} \end{aligned} \tag{8.40}$$

Then

1. Choose $n = \text{neg}$ or $n = \text{pos}$, each with probability $\frac{1}{2}$.
2. Choose $a \in s_n$ according to Q_n . \diamond

Partitioned sampling has two weaknesses. First, it requires $P(s_n)$ to be easy to compute for all $n \in N$. 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 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 N , and according to a different distribution over part s_n for each $n \in N$. Doing so and returning weighted samples to adjust for the differences in distribution comprises ***partitioned importance sampling***.

First, to restrict a probability measure P to a measurable set A *without* renormalizing it, we define

$$\begin{aligned} \text{subcond} : \text{Set } X \rightharpoonup [0, 1] &\Rightarrow \text{Set } X \Rightarrow \text{Set } X \rightharpoonup [0, 1] \\ \text{subcond } P A &:= \lambda A' \in \text{domain } P. P(A' \cap A) \end{aligned} \tag{8.41}$$

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 X, P .
- An at-most-countable index set N .
- A probability mass function $p : N \rightarrow [0, 1]$ such that $p_n > 0$ for all $n \in N$.
- A partition $s : N \rightarrow \text{Set } X$ of X into $|N|$ measurable parts.
- Candidate probability measures $Q : N \rightarrow \text{Set } X \rightharpoonup [0, 1]$, one for each partition.

To sample from X according to P ,

1. Choose $n \in N$ with probability $p(n)$.
2. Choose $a \in X$ according to $Q(n)$.
3. Compute $w := \frac{1}{p(n)} \cdot \text{diff}^+(\text{subcond } P(s(n)))(Q(n))a$.
4. Return the weighted sample $\langle a, w \rangle$.

The function $\text{diff}^+(\text{subcond } P(s(n)))(Q(n))$, with type $X \rightarrow [0, +\infty)$, is a **Radon-Nikodým³ derivative**.

If P has density f , $Q(n)$ has density g , and $a \in s(n)$ implies $g(a) > 0$, then⁴

$$\text{diff}^+(\text{subcond } P(s(n)))(Q(n))a = \begin{cases} f(a) / g(a) & \text{if } a \in s(n) \\ 0 & \text{otherwise} \end{cases} \quad (8.42)$$

Appendix B has formal definitions and more details. We use 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 equal to the true expected values. This is true of partitioned importance sampling under reasonable conditions, which are analogous to the support of $\text{subcond } P(s(n))$ being no larger than that of $Q(n)$. The formal statement of the theorem and its proof are in Appendix B.

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,

- $X := [0, 1] \times [0, 1]$ and P is the uniform measure on X (i.e. area).
- $N := \{\text{left}, \text{right}\}$.
- $p := [\text{left} \mapsto 0.4, \text{right} \mapsto 0.6]$.

³Pronounced “RADOn neekohDIM,” and named after Austrian mathematician Johann Radon and Polish mathematician Otto Nikodým.

⁴The equality in (8.42) holds $(Q(n))$ -almost everywhere.

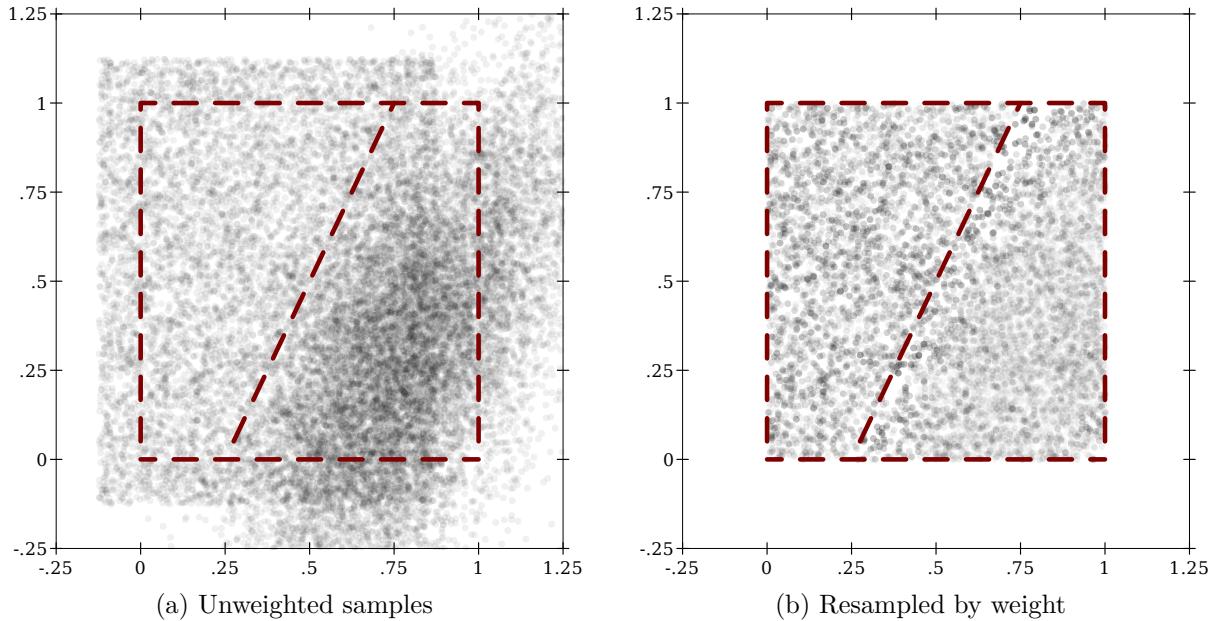


Figure 8.13: Partitioned importance sampling used to sample uniformly in a partition of the unit square, using two overlapping candidate distributions.

- $s_{\text{left}} = \{\langle x, y \rangle \in X \mid y > 2 \cdot x - \frac{1}{2}\}$, similarly for s_{right} .
- Q_{left} is the uniform measure on a superset of s_{left} , and 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 $f : \mathbb{R} \times \mathbb{R} \Rightarrow [0, +\infty)$ to represent P .
- A family of predicates $s? : N \Rightarrow X \Rightarrow \text{Bool}$ to decide $a \in s n$.
- Candidate densities $g : N \Rightarrow X \Rightarrow [0, +\infty)$ to represent Q .

It computes weights using $\text{diff}^+(\text{subcond } P(s n)) (Q n) a = f a / g n a$ when $s? n a = \text{true}$.

It directly represents only N and p , but we will find even this to be infeasible shortly. ◇

Two properties make the preceding example simple. First, the partition has finitely many parts. Second, the measures $\text{subcond } P(s n)$ and $Q n$ have densities, which ensures $\text{diff}^+(\text{subcond } P(s n)) (Q n)$ 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 $\Omega := J \rightarrow [0, 1]$ 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 it is difficult to prove that any given infinite-dimensional Radon-Nikodým derivative exists.

Fortunately, we can prove they exist if the two measures differ in only finitely many axes. More precisely, let $P_1 : \text{Set } \Omega \rightarrow [0, 1]$ and $P_2 : \text{Set } \Omega \rightarrow [0, 1]$ be probability measures, and $J' \subseteq J$ be a finite set of tree indexes. Suppose P_1 can be factored into a distribution P'_1 over finite prefixes $J' \rightarrow [0, 1]$ and a distribution over suffixes $(J \setminus J') \rightarrow [0, 1]$, and P'_2 can be similarly factored into P'_2 and *the same* distribution over suffixes. Then, under reasonable conditions (which are analogous to the support of P'_1 being no larger than that of P'_2), $\text{diff}^+ P_1 P_2$ exists and can be computed using $\text{diff}^+ P'_1 P'_2$. Appendix B contains a formal statement and proof.

To ensure $\text{subcond } P(s n)$ and $Q n$ differ in only finitely many axes, we partition Ω according to branch traces. Each branch trace corresponds with a program that reads any $\omega \in \Omega$ at only finitely many indexes $J' \subseteq J$. [XXX: connect idea better]

In the remainder of this subsection, assume a fixed program p . Let $f := \llbracket p \rrbracket_{\perp^*}^\Downarrow : \langle\langle \Omega, T \rangle, \langle \rangle \rangle \rightsquigarrow_{\perp^*} Y$ be its interpretation as a bottom* arrow computation, with maximal domain A^* . Define $T^* := \text{image } (\text{fst} \ggg \text{snd}) A^*$ as its **maximal branch trace set** and $\Omega^* := \text{image } (\text{fst} \ggg \text{fst}) A^*$ as its **maximal random source set**.

We need a notion of the random sources that agree with a given branch trace $t \in T$; i.e. those $\omega \in \Omega$ for which $f \langle\langle \omega, t \rangle, \langle \rangle \rangle \neq \perp$.

Definition 8.27 (induced random sources). *Let $t \in T$ be a branch trace. The **random sources induced by t** are a subset of Ω defined by $\Omega' := \{\omega \in \Omega \mid f \langle\langle \omega, t \rangle, \langle \rangle \rangle \neq \perp\}$.*

Equivalently, Ω' is the set of $\omega \in \Omega$ for which $\langle\langle \omega, t \rangle, \langle \rangle \rangle \in A^*$.

[XXX: graph of induced partition from program if (random < random) true false?]

Theorem 8.28. Let $t \in T$ induce Ω' . $\Omega' \neq \emptyset$ if and only if $t \in T^*$.

Proof. By definition of T^* , $\Omega' \neq \emptyset$ if and only if there is an $\omega \in \Omega'$ such that $\langle\langle\omega, t\rangle, \langle\rangle\rangle \in A^*$. \square

Using 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 T or T^* as the partition index set because many branch traces can induce the same random sources.

Example 8.29. For the program

$$\text{if (random} < p) 0 \text{ random} \quad (8.43)$$

there are at least two branch traces in the program's maximal domain: $t_0 := [j_0 \mapsto \text{true}, * \mapsto \perp]$ and $t_1 := [j_0 \mapsto \text{false}, * \mapsto \perp]$. There is also $[j_0 \mapsto \text{false}, \text{left } j_0 \mapsto \text{true}, * \mapsto \perp]$, because it agrees with every execution that $[j_0 \mapsto \text{false}, * \mapsto \perp]$ agrees with. In fact, there are infinitely many branch traces in T^* that induce the same random sources as either t_0 or t_1 . [XXX: tree diagrams?] \diamond

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- \perp elements. Two branch traces that differ by returning respectively `true` and `false` for the same j may represent different execution paths, so they must be incomparable.

Definition 8.30 (branch trace partial order). $t_1 \leq t_2$ when for all $j \in J$, $t_1 j = \perp$ or $t_1 j = t_2 j$.

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:

$$\begin{aligned}
 \text{trace-inf} : \text{Set } T &\Rightarrow T \\
 \text{trace-inf } T' := \lambda j \in J. \text{case} & \quad \text{proj } j \, T' \\
 & \quad \{b\} \longrightarrow b \\
 & \quad \text{else} \longrightarrow \perp
 \end{aligned} \tag{8.44}$$

Theorem 8.31 (trace infimum). *Let $T' \subseteq T$ and $t_* := \text{trace-inf } T'$. Then*

- *For all $t' \in T'$, $t_* \leq t'$.*
- *For all $t \in T$, if for all $t' \in T'$, $t \leq t'$, then $t \leq t_*$.*

Proof. Let $t' \in T'$ and $j \in J$. If $\text{proj } j \, T' = \{b\}$ for $b \in \text{Bool}_\perp$, then $t_* \, j = t' \, j = b$. Otherwise, $t_* \, j = \perp$. Thus $t_* \leq t'$.

Let $t \in T$ and suppose that for all $t' \in T'$, $t \leq t'$. Let $j \in J$. If $\text{proj } j \, T' = \{b\}$, then there are two cases: $t \, j = \perp$, or $t \, j = t_* \, j = b$. Otherwise there exists a $t' \in T'$ such that $t \, j \neq t' \, j$, so $t \, j = \perp$. Thus $t \leq t_*$. \square

Any two comparable traces in T^* induce the same random sources.

Theorem 8.32 (comparable implies equivalent). *Let $t_1, t_2 \in T^*$ induce Ω_1, Ω_2 . If $t_1 \leq t_2$ or $t_2 \leq t_1$, then $\Omega_1 = \Omega_2$.*

Proof. It suffices to consider $t_1 \leq t_2$; the $t_2 \leq t_1$ case follows from reflexivity of $(=)$.

Suppose $\omega \in \Omega_1$, so $f \langle \langle \omega, t_1 \rangle, \langle \rangle \rangle \neq \perp$. Let $J' \subseteq J$ such that $j \in J'$ if and only if $\langle \omega, t_1 \rangle$ agrees with the $\text{ifte}_{\perp^*}^\psi$ subcomputation at index j . For all $j \in J'$, $t_1 \, j \neq \perp$, so $t_2 \, j = t_1 \, j$ by definition of (\leq) . Therefore, $\langle \omega, t_2 \rangle$ also agrees with every $\text{ifte}_{\perp^*}^\psi$ subcomputation at every index $j \in J'$, so $f \langle \langle \omega, t_2 \rangle, \langle \rangle \rangle \neq \perp$. Therefore $\omega \in \Omega_2$, so $\Omega_1 \subseteq \Omega_2$.

Suppose $\omega \notin \Omega_1$. Let $J' \subseteq J$ such that $j \in J'$ if and only if $t_1 \, j \neq \perp$. Because $f \langle \langle \omega, t_1 \rangle, \langle \rangle \rangle = \perp$, there exists a $j \in J'$ such that the $\text{ifte}_{\perp^*}^\psi$ subcomputation at index j disagrees with $\langle \omega, t_1 \rangle$. Because $t_1 \, j = t_2 \, j$ by definition of (\leq) , the $\text{ifte}_{\perp^*}^\psi$ subcomputation at index j also disagrees with $\langle \omega, t_2 \rangle$, so $f \langle \langle \omega, t_2 \rangle, \langle \rangle \rangle = \perp$. Therefore $\omega \notin \Omega_2$, so $\Omega_2 \subseteq \Omega_1$. \square

Corollary 8.33 (infimum in T^* implies equivalent). *Let $t_1, t_2 \in T^*$ induce Ω_1, Ω_2 , and define $t_* := \text{trace-inf } \{t_1, t_2\}$. If $t_* \in T^*$, then $\Omega_1 = \Omega_2$.*

If 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 $t \in T^*$ induce Ω' , T' be the largest subset of T^* that induces Ω' , and $t_* := \text{trace-inf } T'$. Then $t_* \in T'$.*

Proof. Let $\omega \in \Omega'$. By definition of **trace-inf**, every $\text{ifte}_{\perp^*}^\Downarrow$ subcomputation agrees with $\langle \omega, t_* \rangle$. Therefore $f \langle \langle \omega, t_* \rangle, \langle \rangle \rangle \neq \perp$, so t_* induces Ω' . \square

Theorem 8.35 (infimum not in T^* implies disjoint). *Let $t_1, t_2 \in T^*$ induce Ω_1, Ω_2 , and define $t_* := \text{trace-inf } \{t_1, t_2\}$. If $t_* \notin T^*$, then $\Omega_1 \cap \Omega_2 = \emptyset$.*

Proof. Let T_1, T_2 be the largest subsets of T^* that induce Ω_1, Ω_2 . Let $t_{1*} := \text{trace-inf } T_1$ and $t_{2*} := \text{trace-inf } T_2$. Because $t_* \notin T^*$, $t_{1*} \neq t_{2*}$.

Let $\omega \in \Omega_1$. For every $j \in J$ for which $t_{1*} j \neq \perp$, there is an $\text{ifte}_{\perp^*}^\Downarrow$ subcomputation at index j that agrees with $\langle \omega, t_{1*} \rangle$. But because $t_{2*} \neq t_{1*}$, there exists a $j \in J$ for which an $\text{ifte}_{\perp^*}^\Downarrow$ subcomputation at index j disagrees with $\langle \omega, t_{2*} \rangle$. Therefore $\omega \notin \Omega_2$. By a symmetric argument, $\omega \in \Omega_2$ implies $\omega \notin \Omega_1$. \square

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** T_* is the set of minimal elements in T^* , or*

$$T_* := \{t_1 \in T^* \mid \forall t_2 \in T^*. t_2 \leq t_1 \implies t_2 = t_1\} \quad (8.45)$$

Theorem 8.37. T_* induces Ω^* .

Proof. Let $t \in T^*$ induce Ω' and T' be the largest subset of T^* that induces Ω' . Its minimum $t_* := \text{trace-inf } T'$ is in T_* . \square

Theorem 8.38 (T_* partitions). *Let $t_1, t_2 \in T_*$ induce respectively Ω_1 and Ω_2 . If $t_1 \neq t_2$, then $\Omega_1 \cap \Omega_2 = \emptyset$.*

Proof. Let $t_* := \text{trace-inf } \{t_1, t_2\}$. Because t_1 and t_2 are minimal, $t_* \notin T^*$. By Theorem XXX, $\Omega_1 \cap \Omega_2 = \emptyset$. \square

We can thus sample a partition index $t \in T_*$, which induces a unique part from a partition of Ω^* .

A program’s minimal branch trace set T_* contains only the actual branches taken when running the program on every $\omega \in \Omega^*$. Therefore, one way to sample from 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 ω 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 Ω^* subsets they induce, are easy to compute. Second, we are interested in sampling from an *arbitrarily low-probability subset* of Ω^* , which may be covered by the partition induced by an *arbitrarily low-probability subset* of T_* .

It appears we have a chicken-and-egg problem, in that

1. Sampling in a small subset of Ω^* requires sampling in a small subset of T_* .
2. Sampling in a small subset of T_* requires sampling in a small subset of Ω^* .

Fortunately, if we allow ourselves subsets of a larger set than T_* , and allow ourselves to sample within overapproximating covers of Ω^* subsets, we can use approximate preimage computation to sample from T_* and Ω^* subsets simultaneously.

8.4.3 Approximate Partitions of Probabilistic Program Domains

The idea is to define a set of branch traces T_+ that is derived only from a program’s shape, not its actual executions. We ensure that $T_* \subseteq T_+$, and that every $t \in T_+ \setminus T_*$ induces \emptyset , so that T_+ induces the same partition as T_* . We define an algorithm for sampling from T_+ , which does not require running a probabilistic program on any $\omega \in \Omega$. We then extend this

$\text{Idxs} ::= \langle \rangle \mid \langle \text{Idxs}, \text{Idxs} \rangle$	$\text{ifte}_{\text{idxs}^*} : (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs})$
$\mid \text{if-idxs } \text{J} (\text{1} \Rightarrow \text{Idxs}) (\text{1} \Rightarrow \text{Idxs})$	$\text{ifte}_{\text{idxs}^*} \ i_1 \ i_2 \ i_3 \ j := \text{let } \text{idxs}_2 := \lambda 0. i_2 \ (\text{left } (\text{right } j))$
$\text{arr}_{\text{idxs}^*} : (x \Rightarrow y) \Rightarrow (\text{J} \Rightarrow \text{Idxs})$	$\text{idxs}_3 := \lambda 0. i_3 \ (\text{right } (\text{right } j))$
$\text{arr}_{\text{idxs}^*} f j := \langle \rangle$	$\text{in } \langle i_1 \ j, \text{if-idxs } j \ \text{idxs}_2 \ \text{idxs}_3 \rangle$
$(\ggg_{\text{idxs}^*}) : (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs})$	$\text{lazy}_{\text{idxs}^*} : (1 \Rightarrow (\text{J} \Rightarrow \text{Idxs})) \Rightarrow (\text{J} \Rightarrow \text{Idxs})$
$(i_1 \ggg_{\text{idxs}^*} i_2) j := \langle i_1 \ (\text{left } j), i_2 \ (\text{right } j) \rangle$	$\text{lazy}_{\text{idxs}^*} i \ j := i \ 0 \ j$
$(\&&_{\text{idxs}^*}) : (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs})$	$\text{random}_{\text{idxs}^*} : \text{J} \Rightarrow \text{Idxs}$
$(i_1 \&&_{\text{idxs}^*} i_2) j := \langle i_1 \ (\text{left } j), i_2 \ (\text{right } j) \rangle$	$\text{random}_{\text{idxs}^*} j := \langle \rangle$

(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 λ -calculus.

```

sample-traces : Idxs → ⟨ℝ, Rect T⟩
sample-traces idxs := sample-traces* idxs ⟨1, T⟩

sample-traces* : Idxs ⇒ ⟨ℝ, Rect T⟩ ⇒ ⟨ℝ, Rect T⟩
sample-traces* ⟨⟩ pt := pt
sample-traces* ⟨idxs₁, idxs₂⟩ pt := let pt' := sample-traces* idxs₁ pt
                                         in sample-traces* idxs₂ pt'
sample-traces* (if-idxs j idxs₂ idxs₃) ⟨pt, T'⟩ := let ⟨pb, b⟩ := sample-branch Bool⊥
                                                       pt' := ⟨pt · pb, unproj j T' {b}⟩
                                                       in case b
                                                         true → sample-traces* (idxs₂ 0) pt'
                                                         false → sample-traces* (idxs₃ 0) pt'
                                                         ⊥ → pt'

```

(b) The stochastic function `sample-traces` samples a T' , and returns T' and its probability.

Figure 8.14: Branch index collecting semantics.

algorithm to use preimage computation to sample in arbitrarily good approximations of small subsets of T_* and Ω^* .

Defining T_+ in terms of a program's branching shape requires an additional abstract interpretation. Figure 8.14a defines the *indexes arrow*. Its type is $\text{J} \Rightarrow \text{Idxs}$, 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 $\lambda j. \langle \rangle$, which takes the function's computation index and returns no decisions. Composition and pairing of subcomputations i_1 and i_2 both return $\langle i_1 \ (\text{left } j), i_2 \ (\text{right } j) \rangle$: a

node with two children that contain the feasible branch decisions in their subcomputations.

Only $\text{ifte}_{\text{idxs}^*}$ does more than simple structural recursion: it returns $\text{if-idxs } j \text{ idxs}_2 \text{ idxs}_3$ to represent a decision at computation index j . The children idxs_2 and idxs_3 are lazy, abstract representations of the 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 $\text{idxs} : \text{Idxs}$. It returns a pair $\langle p_t, T' \rangle$, where T' is the sampled rectangle and p_t is the probability with which it was chosen. It assumes a stochastic procedure $\text{sample-branch} : \text{Set Bool}_\perp \Rightarrow \langle \mathbb{R}, \text{Bool}_\perp \rangle$, where $\text{sample-branch } B$ returns any member of B with some nonzero, constant probability. At index j , for the branch choice $\langle p_b, b \rangle := \text{sample-branch } \text{Bool}_\perp$, T' is restricted using $\text{unproj } j \text{ T}' \{b\}$.

Although sample-traces returns rectangles, it is easy to transform one into a single trace using trace-inf ; i.e. $\text{trace-inf} (\text{snd} (\text{sample-traces} \text{ idxs}))$ samples a branch trace.

Let $\text{idxs} := \llbracket p \rrbracket_{\text{idxs}^*}^\downarrow j_0$.

Definition 8.39 (feasible branch traces). *The **feasible branch traces** T_+ are those $t \in T$ for which $\Pr[t = \text{trace-inf} (\text{snd} (\text{sample-traces} \text{ idxs}))] > 0$.*

Because sample-traces^* imposes a total order on evaluation, any terminating application of it induces a total order on the indexes j in applications matching $\text{if-idxs } j \text{ idxs}_2 \text{ idxs}_3$. Let j_1, j_2, \dots, j_n be those indexes, with corresponding branch choices b_1, b_2, \dots, b_n . Define T'_1, T'_2, \dots, T'_n by $T'_0 := T$ and $T'_i := \text{unproj } j_i \text{ T}'_{i-1} \{b_i\}$.

Theorem 8.40 (sample-traces soundness). $T_* \subseteq T_+$.

Proof. Let $t \in T_*$. It suffices to show that there exists an n and a sequence of branch choices b_1, b_2, \dots, b_n for which $t = \text{trace-inf } T'_n$.

First, we prove by induction the seemingly weaker statement that there exist n and branch choices for which $t \in T'_n$. Let j_1, j_2, \dots, j_n be the in-order indexes at which $t \mid_j \neq \perp$. Clearly $t \in T'_0 = T$. If $t \in T'_{i-1}$, then $b_i := t \mid_{j_i}$ implies $t \in T'_i = \text{unproj } j_i \text{ T}'_{i-1} \{b_i\}$.

For any $j \in \{j_1, j_2, \dots, j_n\}$, $\{t|j\} = \text{proj } j|T'_n$. For any other j , $t|j = \perp$ and $\text{proj } j|T'_n = \text{Bool}_\perp$. By definition of **trace-inf**, therefore $t = \text{trace-inf } T'_n$. \square

For T_+ to induce a partition, every $t \in T_+ \setminus T_*$ must induce \emptyset .

Theorem 8.41 (**sample-traces** non- \emptyset -unique). *For all $t \in T_+$, if $t \notin T_*$ then t induces \emptyset .*

Proof. Let j_1, j_2, \dots, j_n and b_1, b_2, \dots, b_n for a terminating evaluation of **sample-traces*** `idxs` $\langle 1, T \rangle$.

Suppose $T'_n \cap T_* = \emptyset$. Then there exists an i such that $T'_{i-1} \cap T_* \neq \emptyset$ and $T'_i \cap T_* = \emptyset$.

Thus $b_i \notin \text{proj } j_i|T_*$, so f does not agree with any $t \in T'_i$.

Let $t := \text{trace-inf } T'_n$, which by definition of **trace-inf** and **sample-traces*** is in T'_n . Because $T'_n \subseteq T'_i$, f does not agree with t , so t induces \emptyset . \square

Corollary 8.42 (**sample-traces** partitioning). *T_+ induces a partition of Ω^* .*

To be used in partitioned importance sampling, the probability returned by **sample-traces** must be correct.

Theorem 8.43 (**sample-traces** correctness). *If $\langle p'_t, T' \rangle := \text{sample-traces } \text{idxs}$, then $\Pr[T'] = p'_t$.*

Proof. Let $p_{b_1}, p_{b_2}, \dots, p_{b_n}$ be the probabilities returned from **sample-branch** for b_1, b_2, \dots, b_n . The probability of b_1, b_2, \dots, b_n is thus $p'_t := p_{b_1} \cdot p_{b_2} \cdot \dots \cdot p_{b_n}$. Because the transformation from b_1, b_2, \dots, b_n to T'_n is injective, $\Pr[T'_n] = p'_t$. \square

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 b_i , there is a nonzero probability that $b_i = \perp$, which is a recursion base case. \square

We finally have a way to use partitioned importance sampling to sample within the preimage of some set B . Define

$$f := \llbracket p \rrbracket_{\perp^*}^\psi j_0 \quad h' := \llbracket p \rrbracket_{\text{pre}^*}^\psi j_0 \quad \text{idxs} := \llbracket p \rrbracket_{\text{idxs}^*}^\psi j_0 \quad (8.46)$$

$\text{Idxs} ::= \langle \rangle \mid \langle \text{Idxs}, \text{Idxs} \rangle$	$\text{ifte}_{\text{idxs}^*} : (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs})$
$\text{if-idxs J } (1 \Rightarrow \text{Idxs}) (1 \Rightarrow \text{Idxs})$	$\text{ifte}_{\text{idxs}^*} \ i_1 \ i_2 \ i_3 \ j := \text{let } \text{idxs}_2 := \lambda 0. i_2 \ (\text{left } (\text{right } j))$
random-idxs J	$\text{idxs}_3 := \lambda 0. i_3 \ (\text{right } (\text{right } j))$
	in $\langle i_1 \ j, \text{if-idxs } j \ \text{idxs}_2 \ \text{idxs}_3 \rangle$
$\text{arr}_{\text{idxs}^*} : (x \Rightarrow y) \Rightarrow (\text{J} \Rightarrow \text{Idxs})$	$\text{lazy}_{\text{idxs}^*} : (1 \Rightarrow (\text{J} \Rightarrow \text{Idxs})) \Rightarrow (\text{J} \Rightarrow \text{Idxs})$
$\text{arr}_{\text{idxs}^*} \ f \ j := \langle \rangle$	$\text{lazy}_{\text{idxs}^*} \ i \ j := i \ 0 \ j$
$(\ggg_{\text{idxs}^*}) : (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs})$	$\text{random}_{\text{idxs}^*} : \text{J} \Rightarrow \text{Idxs}$
$(i_1 \ggg_{\text{idxs}^*} i_2) \ j := \langle i_1 \ (\text{left } j), i_2 \ (\text{right } j) \rangle$	$\text{random}_{\text{idxs}^*} \ j := \text{random-idxs } j$
$(\&&_{\text{idxs}^*}) : (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs}) \Rightarrow (\text{J} \Rightarrow \text{Idxs})$	
$(i_1 \&&_{\text{idxs}^*} i_2) \ j := \langle i_1 \ (\text{left } j), i_2 \ (\text{right } j) \rangle$	

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 $\text{refine } A := \text{ap}'_{\text{pre}^*} (h' A) B$. Then

1. Let $\langle p_t, T' \rangle := \text{sample-traces idxs}$.
2. Let $t := \text{trace-inf } T'$.
3. Let $A' := \text{refine } ((\Omega \times \{t\}) \times \{\langle \rangle\})$.
4. Let $\Omega' := \text{image } (\text{fst} \ggg \text{fst}) A'$. If $\Omega' = \emptyset$, reject.
5. Choose $\omega \in \Omega'$ according to $Q t$. If $f \langle \langle \omega, t \rangle, \langle \rangle \rangle \notin B$, reject.
6. Compute weight $w := \frac{1}{p_t} \cdot \text{diff}^+ (\text{subcond } P \ \Omega'') (Q t) \ \omega$, where Ω'' is the set of random sources induced by t .

Computing $\text{diff}^+ (\text{subcond } P \ \Omega'') (Q t)$ does not require Ω'' , as we will demonstrate shortly.

Samples are rejected for two reasons. The first is when $\Omega' = \emptyset$ 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 Ω 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

$$f := \llbracket p \rrbracket_{\perp^*}^\psi j_0 \quad h' := \llbracket p \rrbracket_{\text{pre}^*}^\psi j_0 \quad \text{idxs} := \llbracket p \rrbracket_{\text{idxs}^*}^\psi j_0$$

where $f : \text{Rect } \langle\langle \Omega, T \rangle\rangle \rightsquigarrow_{\perp} Y$

```

refine : Rect <Ω, T> ⇒ Rect <Ω, T>
refine A := image fst (ap'_{\text{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 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 $\text{random}_{\text{idxs}^*} j := \text{random-idxs } j$ instead of $\text{random}_{\text{idxs}^*} j := \langle \rangle$.

The proofs of the preceding theorems indicate the properties the new partition sampler must have.

- It must be sound: for any $t \in T_*$, with positive probability, it constructs a T' whose infimum is t .
- It must partition: if it constructs a T' whose infimum is not in T_* , T' must induce \emptyset .
- It must terminate: branch choices must be \perp 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 $\Omega \times T$ instead of from a rectangular partition of T .
- For random-idxs j , it uses `sample-real-part` to sample from a partition of $\text{proj } j \Omega'$.
- It uses `refine` to shrink the part's covering rectangle after every choice.
- It stops immediately if it receives \emptyset , which `refine` may return.
- It chooses branches from $\text{proj } j T' \cup \{\perp\}$ instead of Bool_\perp , which allows `refine` to rule out branch choices that disagree with Ω' .

We assume that for each input, $\text{sample-real-part} : \text{Rect } \mathbb{R} \Rightarrow \langle \mathbb{R}, \text{Rect } \mathbb{R} \rangle$ 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 \emptyset , it returns \perp weighted by 0. If the part is nonempty, it samples from the random sources and weights the sample. For sample ω and trace t , if $f \langle \langle \omega, t \rangle, \langle \rangle \rangle \notin B$ then $\langle \omega, t \rangle$ is not in the preimage of B , so it weights ω by 0, which is equivalent to rejecting it.

If the sample's image is in B , `sample-preimage` computes the sample's weight as $1/p_n \cdot 1/q_\omega$. To correctly do partitioned importance sampling, the weight should be $1/p_n \cdot \text{diff}^+ (\text{subcond } P \Omega'') (Q n) \omega$, where p_n is the probability of choosing the part. The leading term is thus correct, so we need to show $1/q_\omega = \text{diff}^+ (\text{subcond } P \Omega'') (Q n) \omega$.

To state the theorem, we need some definitions. Let $h := \llbracket p \rrbracket_{\text{pre}*}^\psi j_0$ be the interpretation of p as a preimage arrow computation, and $\Omega'' := \text{image} (\text{fst} \ggg \text{fst}) (\text{ap}_{\text{pre}} (h (\Omega' \times T') \times \{\langle \rangle\}) B)$

be the exact part under its covering rectangle Ω' . Let $J' \subseteq J$ such that $j \in J'$ if and only if `sample-part` is applied to `random-idxs 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 n be the partition index of the part covered by $\Omega' \times T'$.

Theorem 8.45. *Let $Q|n$ have a density q when restricted to indexes J' and be uniform on $J \setminus J'$. Suppose `sample-source` $(\Omega' \times T')$ chooses $\omega \in \Omega'$ according to $Q|n$. If $\omega \in \Omega''$, then $\text{diff}^+ (\text{subcond } P|\Omega'') (Q|n) \omega = 1/(q(\text{restrict } \omega|J'))$.*

Proof. This is a straightforward application of Theorem B.25, so we need only meet the conditions. Because J' is finite,

- The subprobability measure `subcond P|Ω''` can be factored into $P' : \text{Set}(J' \rightarrow [0, 1]) \Rightarrow [0, 1]$ and a uniform probability measure on $J \setminus J' \rightarrow [0, 1]$.
- The probability measure $Q|n$ can be factored into $Q' : \text{Set}(J' \rightarrow [0, 1]) \Rightarrow [0, 1]$ and the same uniform probability measure.

A density for P' that is uniform on Ω'' is

$$\begin{aligned} p : (J' \rightarrow [0, 1]) &\rightarrow [0, +\infty) \\ p \omega &:= \begin{cases} 1 & \text{if } (\omega \in \Omega'') \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{8.47}$$

By assumption, the density of Q' is $q : (J' \rightarrow [0, 1]) \rightarrow [0, +\infty)$. Therefore, if $\omega \in \Omega''$,

$$\text{diff}^+ (\text{subcond } P|\Omega'') (Q|n) \omega = \frac{p(\text{restrict } \omega|J')}{q(\text{restrict } \omega|J')} = \frac{1}{q(\text{restrict } \omega|J')} \tag{8.48}$$

□

Thus, if `sample-source` $(\Omega' \times T') = \langle q(\text{restrict } \omega|J'), \omega \rangle$, 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 Ω' . Let $m : \text{Set} \rightarrow [0, +\infty]$

be Lebesgue measure on \mathbb{R} . Define

```
sample-source : Rect <math>\langle \Omega, T \rangle \Rightarrow \langle \mathbb{R}, \Omega \rangle</math>
sample-source (<math>\Omega' \times T'\>) := let <math>q_\omega := 1 / \prod_{j \in J} m(\text{proj } j \Omega')</math>
                                         <math>\omega := \lambda j \in J. \text{sample-uniform } (\text{proj } j \Omega')</math>
                                         in <math>\langle q_\omega, \omega \rangle</math>
```

(8.49)

Because $\text{Rect } \langle \Omega, T \rangle$ is defined so that only finitely many axes of Ω' are strict subsets of $[0, 1]$, q_ω is well-defined whenever the volume of Ω' is nonzero.⁵ In particular, $m(\text{proj } j \Omega') < 1$ if $j \in J'$, otherwise 1.

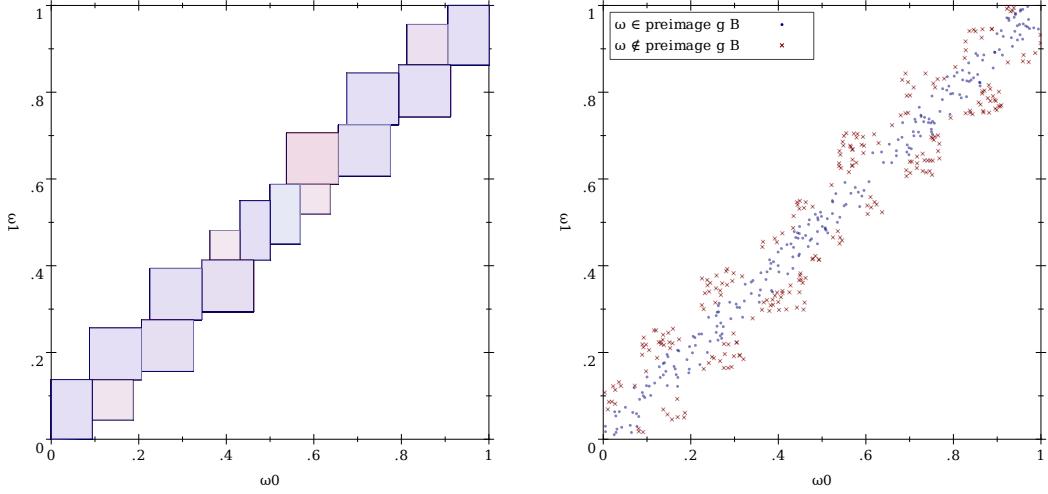
An implementation of `sample-source` cannot compute a product over all J , nor construct a mapping with domain J . The representations of $\text{Rect } \Omega$ and $\omega \in \Omega$ given in Figures 8.5 and 8.6 make getting around this easy. The function `omega-set-sample` in Figure 8.6 implements $\lambda j \in J. \text{sample-uniform } (\text{proj } j \Omega')$ 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 $\prod_{j \in J} m(\text{proj } j \Omega')$ to measure the volumes of Ω 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 8.17a shows the result of sampling within the preimage of $[-0.05, 0.05]$ under the interpretation of this program as a random variable:

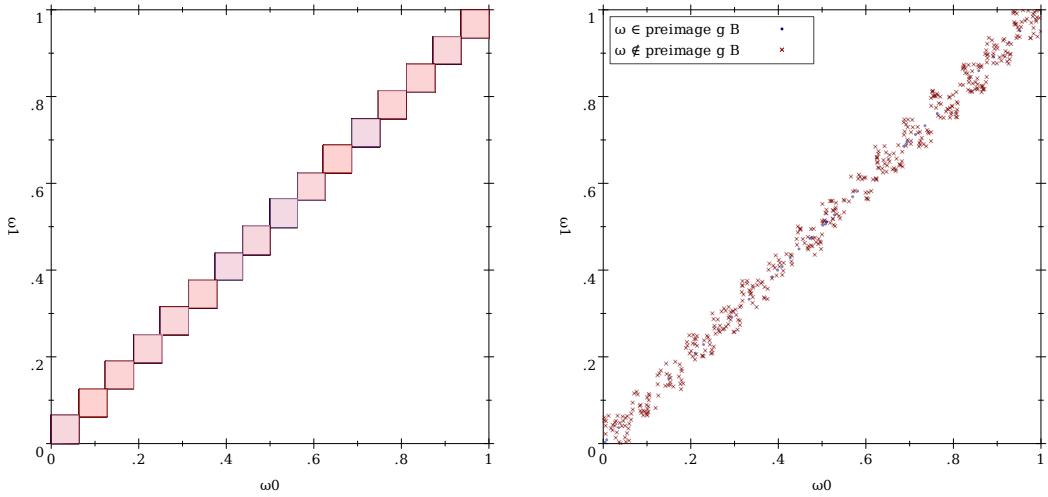
```
(define/drabayes diagonal
  (let ([x (random)]
        [y (random)])
    (- y x)))
```

The left plot shows the results returned by the implementation of `sample-part`: sampled parts from a rectangle covering the true preimage, which surrounds the line $\omega_1 = \omega_0$. (There are many duplicates.) The right plot shows the result of sampling once within each part uniformly; in this case, 244 out of 500 samples are inside the preimage set.

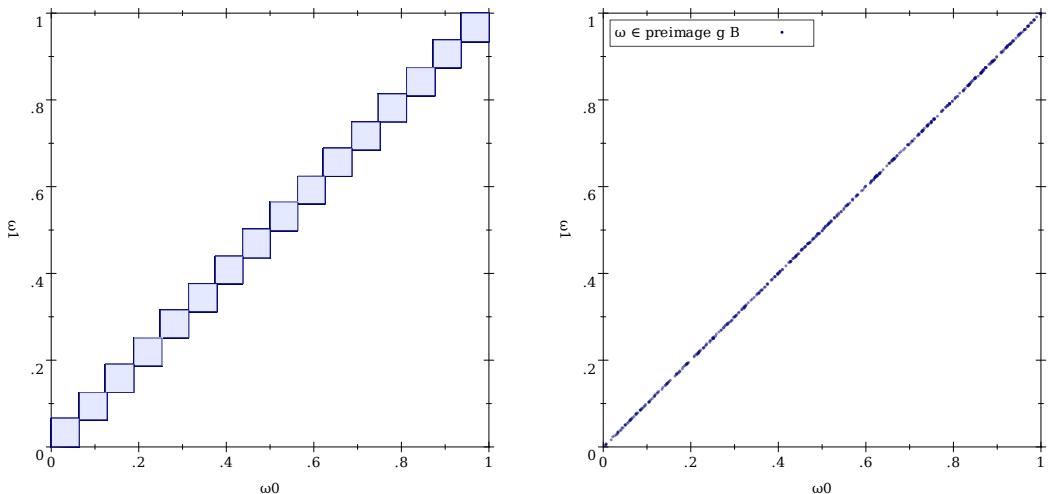
⁵In practice, we do not have to consider this case. The implementation of `sample-source` may return reciprocal densities, so it returns the volume of Ω' , which is always well-defined.



(a) Results of **sample-part** and **sample-source** for the preimage of $[-0.05, 0.05]$. Samples accepted: 244.



(b) Results of **sample-part** and **sample-source** for the preimage of $[-0.002, 0.002]$. Samples accepted: 30.



(c) Results of **sample-part** and **sample-source*** for the preimage of $[-0.002, 0.002]$. Samples accepted: 500.

Figure 8.17: 500 samples using Dr. Bayes's implementations of **sample-part**, **sample-source** and **sample-source***.

Figure 8.17b shows the result of sampling within the preimage of $[-0.002, 0.002]$, for which many fewer samples are in the preimage set; in this case, only 30. In general, for the interpretation of `diagonal`, the proportion of accepted samples in the preimage of $[-\epsilon, \epsilon]$ scales linearly with ϵ . We can mitigate this problem using finer partitions of $\text{proj } j \Omega'$. 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 $\{b\} = [b, b]$ is also a rectangle. To sample within a part, for each axis $j \in J'$, we choose $b \in \text{proj } j \Omega'$, update Ω' using $\text{unproj } j \Omega' \{b\}$, and use `refine` to get better bounds for sampling the other axes.

The following function implements the idea.

```

sample-source* : [J] ⇒ ⟨ℝ, Rect ⟨Ω, T⟩⟩ ⇒ ⟨ℝ, Ω⊥⟩

sample-source* js ⟨qω, ∅⟩           := ⟨0, ⊥⟩
sample-source* ⟨⟩ ⟨qω, Ω' × T'⟩   := let ω := λj ∈ J. sample-uniform (proj j Ω')
                                         in ⟨qω, ω⟩
sample-source* ⟨j, js⟩ ⟨qω, Ω' × T'⟩ := let B := proj j Ω'
                                         b := sample-uniform B
                                         A' := refine (unproj j Ω' {b} × T')
                                         in sample-source* js ⟨qω · 1/(m B), A'⟩

```

(8.50)

Here, $[J]$ is the type of lists of J , or $\langle J, \langle J, \dots \langle J, \langle \rangle \rangle \rangle \rangle$. The caller is expected to linearize J' , the indexes of random values that are actually used while running the program, as $js : [J]$. (Dr. Bayes's implementation of `sample-part` returns `js` in addition to the covering rectangle and its probability.) The density of the sampled ω is computed as $\prod_{j \in J'} 1/(m(\text{proj } j \Omega'_j))$, where Ω'_j are the ever-shrinking inputs to `sample-source*`. Roughly, it is the joint density of *dependent* uniform random variables evaluated at `restrict ω J'`. [XXX: name this dependent sampling?]

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 ω_0 and updating Ω' with $\{\omega_0\}$ allows preimage refinement to determine the range of values for ω_1 for which ω is in the preimage set.

Samples taken using `sample-source*` may still be rejected, when sampling from an

overapproximated projection causes `refine` to return \emptyset . We demonstrate and characterize the conditions under which this happens in Chapter 9.

8.4.5 Self-Adjusting Probabilistic Search

One way to regard `sample-part` is as a search for nonempty sets.

To be correct, `sample-part` must return the actual probabilities with which it chooses each rectangular part cover. The easiest way to do so is to ensure that the transformation from random choices to part covers is injective: that no two combinations of choices result in the same part cover. Then the part cover's probability is the product of the choice probabilities.

This unfortunately rules out backtracking search, because many backtracking paths can result in the same rectangular cover. Fortunately, because we invoke `sample-part` many times, past results can inform future ones.

The main idea is this: instead of just searching by making random choices, build a tree of possible searches. Each child node represents a choice, and parents label their children with probabilities. Leaf nodes contain rectangular part covers. To choose a part cover, repeatedly choose child nodes according to their probabilities. If the resulting leaf's cover is \emptyset , remove it from the tree and adjust the child probabilities. Thus, choice combinations that lead to failure occur at most once. If child probabilities are correctly adjusted, removing a failure leaf does not change the probabilities of successful ones.

Figure 8.18a illustrates the self-adjusting search more generally. Four boxes represent two failure leaves with values x_0 and x_2 , and two success leaves with values x_1 and x_3 . Each leaf value is distinct. Each node has two children, each with some probability, and the child probabilities sum to 1. The dotted outlines show a random path down the search tree ending on a failure leaf x_2 . The probability of this failure is $1 \cdot 1/3 \cdot 4/7 = 4/21$.

To remove it, we must update the entire path back up to the root in a way that maintains the probabilities of every other leaf. Figure 8.18b shows the result of having done so. For example, the probability of x_3 is $1 \cdot 1/3 \cdot 3/7 = 1/7$ in the original tree, and is still

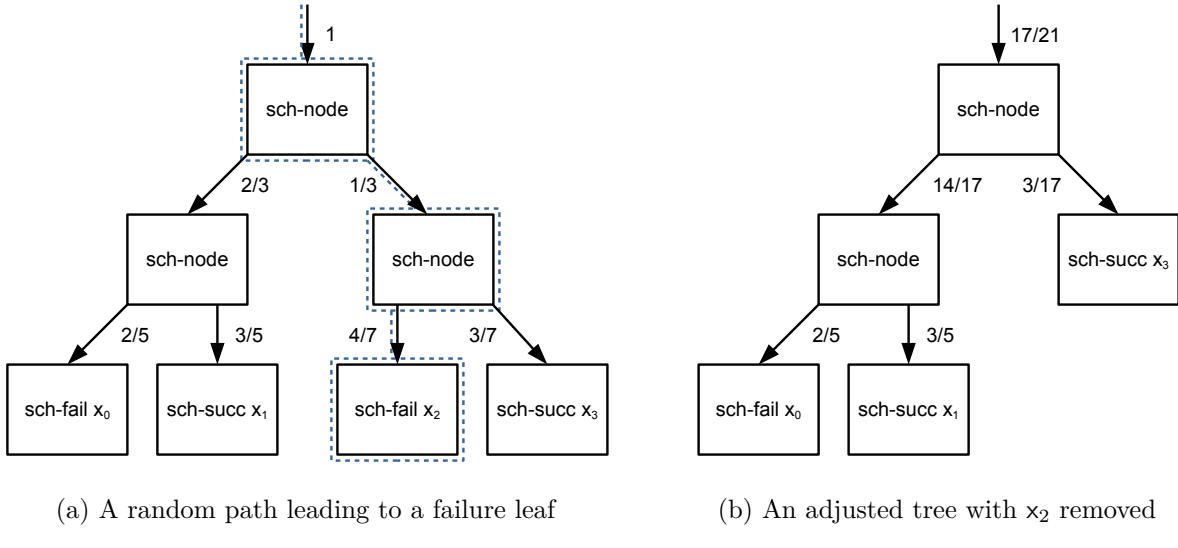


Figure 8.18: A self-adjusting, probabilistic tree search. After $\text{sch-fail } x_2$ is discovered to be a failure node, the path to the root is updated to remove it and maintain the probabilities of the other leaves.

```

Search X ::= sch-fail X
          | sch-succ X
          | sch-node <(0, 1], Search X> <(0, 1], Search X>

adjusted-node : (0, 1] ⇒ <[0, 1], Search X> ⇒ <(0, 1], Search X> ⇒ <(0, 1], Search X>
adjusted-node pt <0, _> <pr, cr> := <pt · pr, cr>
adjusted-node pt <pl, cl> <pr, cr> := let plr := pl + pr
                                         in <pt · plr, sch-node <pl/plr, cl> <pr/plr, cr>>

sample-search : <(0, 1], Search X> ⇒ <<(0, 1], X>, <[0, 1], Search X>>
sample-search <px, sch-succ x> := <<px, x>, <px, sch-succ x>>
sample-search <px, sch-fail x> := <<px, x>, <0, sch-fail x>>
sample-search <pt, sch-node l r> := let sch-node <pl, cl> r' := if (sample-bool (fst l))
                                         (sch-node l r)
                                         (sch-node r l)
                                         <px, <pt', cl'> := sample-search <pt · pl, cl>
                                         pl' := pt' / pt
                                         in <px, adjusted-node pt <pl', cl'> r'

```

Figure 8.19: A data type and algorithm for a self-adjusting, probabilistic tree search.

$17/21 \cdot 3/17 = 1/7$ in the adjusted tree.

Figure 8.19 defines **Search X**, the type of search trees with leaf values **X**. To keep the presentation simple, values with type **Search X** are finite trees. It is easy to extend it to include lazy trees, and not much harder to change **sample-part** to build and update an instance

of $\langle \mathbb{R}, \text{Search}(\text{Rect } \langle \Omega, T \rangle) \rangle$ instead of sampling directly. From here on, we consider only trees in which every pair of child probabilities sums to 1 and every leaf value in the tree is unique.

The `sample-search` function carries out the self-adjusting probabilistic search. It receives a probability and a `Search X` instance; for example, searching the tree in Figure 8.18b is done by `sample-search` $\langle 17/21, \text{sch-node} \dots \rangle$. It returns two pairs: $\langle p_x, x \rangle$, which is the sampled value and its probability, and $\langle p'_t, t' \rangle$, which is the new search tree and its new probability.

We must be sure that p_x is the probability of x .

Theorem 8.46 (`sample-search` returns correct probabilities). *Let $\langle p_t, t \rangle : \langle \mathbb{R}, \text{Search } X \rangle$. Let $\langle \langle p_x, x \rangle, _ \rangle := \text{sample-search } \langle p_t, t \rangle$. If the probability of t is p_t , the probability of x is p_x .*

Proof. By induction on t . Base cases $t = \text{sch-succ } x$ and $t = \text{sch-fail } x$ follow directly from uniqueness of x . For the inductive case $t = \text{sch-node } \langle p_l, c_l \rangle \langle p_r, c_r \rangle$, let `sch-node` $\langle p, c \rangle r' := \text{if } (\text{sample-bool } p_l) \dots$ as in `sample-search`. Because $p_l + p_r = 1$, $\Pr[c = c_l] = p_t \cdot p_l$ and $\Pr[c = c_r] = p_t \cdot p_r$. Apply the inductive hypothesis for cases $c = c_l$ and $c = c_r$. \square

When `sample-search` rebuilds the path from the leaf to the root using `adjusted-node`, we must be sure that `adjusted-node` labels the left and right children with the correct probabilities.

Theorem 8.47 (`adjusted-node` returns correct probabilities). *Let $p_t \in (0, 1]$ and $\langle p'_t, t' \rangle := \text{adjusted-node } p_t \langle p_l, c_l \rangle \langle p_r, c_r \rangle$.*

If $t' = c_r$, then $p'_t = p_t \cdot p_r$.

If $t' = \text{sch-node } \langle p'_l, c_l \rangle \langle p'_r, c_r \rangle$, then $p'_t \cdot p'_l = p_t \cdot p_l$ and $p'_t \cdot p'_r = p_t \cdot p_r$.

Proof. Case $t' = c_r$. Then $p'_t = p_t \cdot p_r$ by definition of `adjusted-node`.

Case $t' = \text{sch-node } \langle p'_l, c_l \rangle \langle p'_r, c_r \rangle$. Let $p_{lr} := p_l + p_r$, so $p'_t \cdot p'_l = (p_t \cdot p_{lr}) \cdot (p_l / p_{lr}) = p_t \cdot p_l$ by the definition of `adjusted-node`, and similarly for $p'_t \cdot p'_r$. \square

Thus, using `adjusted-node` $p_t \langle p'_l, c'_l \rangle \langle p_r, c_r \rangle$ to replace child c_l with c'_l does not affect the probabilities of leaves below c_r . Further, suppose $\langle p_x, \langle p'_t, c'_l \rangle \rangle = \text{sample-search } \langle p_t \cdot p_l, c_l \rangle$ and

$p'_l = p'_t/p_t$ as in `sample-search`. By Theorem 8.47, c'_l will be chosen with probability $p'_l \cdot p_t = p'_t$, as desired.

A simple extension makes trees converge not just to trees without failures, but to trees with stated probabilities for each leaf. Consider the base cases in `sample-search`'s definition:

$$\begin{aligned}\text{sample-search } \langle p_x, \text{sch-succ } x \rangle &:= \langle \langle p_x, x \rangle, \langle p_x, \text{sch-succ } x \rangle \rangle \\ \text{sample-search } \langle p_x, \text{sch-fail } x \rangle &:= \langle \langle p_x, x \rangle, \langle 0, \text{sch-fail } x \rangle \rangle\end{aligned}\tag{8.51}$$

In both cases, returning $\langle p, t \rangle$ in the second of the pair causes the tree to be rebuilt so that x 's probability becomes p . Now define, instead of $\text{sch-succ}, \text{sch-fail} : X \Rightarrow \text{Search } X$, a constructor $\text{sch-leaf} : [0, 1] \Rightarrow X \Rightarrow \text{Search } X$ and

$$\text{sample-search } \langle p_x, \text{sch-leaf } p \ x \rangle := \langle \langle p_x, x \rangle, \langle p, \text{sch-leaf } p \ x \rangle \rangle\tag{8.52}$$

Suppose $t := \text{sch-leaf } p_0 \ x_0$ is a leaf in the tree. Sampling the initial search tree returns x_0 with some probability p_x that is determined by the path to t . With every subsequent tree after x_0 is first returned, sampling returns x_0 with probability p_0 .

A version of `sample-part` that builds such trees using `sch-leaf` might use the actual measure of the part cover Ω' as its probability. Recall, however, that partitioned importance sampling requires `sample-part` to choose parts according to a fixed probability measure. We are fairly certain that partitioned importance sampling is correct even when the distribution over parts varies, as long as it converges pointwise, but we have not yet proved it.

8.5 Conclusions

XXX: segue, possibly give components diagram again

Chapter 9

Example Programs

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth

The correctness proofs in Chapters 7 and 8 assume an idealized model of the host language. While Dr. Bayes's core is nearly a transliteration of the λ_{ZFC} terms that define the approximating semantics, we cannot know how well the theorems apply to Dr. Bayes without testing it.

Besides, we must still demonstrate that it is useful.

9.1 Guaranteed Termination

The theorems in Chapter 7 require a program or expression e to be well-defined (Definition 7.5); in particular, any interpretation $\llbracket e \rrbracket_{\alpha^*}^\psi$ must terminate. The syntax transformers that implement $\llbracket \cdot \rrbracket_{\alpha^*}^\psi$ and Racket's rules for module-level definitions enforce this. For example, the following program is not well-defined according to the semantics:

```
(define/dr bayes (loop) (loop))
```

Further, Racket raises a compile-time error, or more precisely, an expansion-time error. Its interpretation as a bottom* arrow computation is

```
(define loop/bot* (apply/bot* loop/bot* (list)))
```

where `apply/bot*` composes a first-order function's interpretation with a list of argument interpretations. (Here, the list is empty because there are no arguments.) Racket's expander does not allow module-level bindings such as `loop/bot*` to be referenced except by subsequent module-level expressions and expressions in the bodies of lambdas.

On the other hand, this program is well-defined according to the semantics because its recurrences are guarded by `if`:

```
(define/drabayes (loop) (if #t (loop) (loop)))
```

Further, Racket raises no errors. As a bottom* arrow computation, it is

```
(define loop/bot*
  (ifte*/bot* (const/bot* #t)
    (lazy/bot* (delay (apply/bot* loop/bot* (list))))
    (lazy/bot* (delay (apply/bot* loop/bot* (list))))))
```

Racket allows this definition of `loop/bot*` because the inner reference to `loop/bot*` is within a `delay` form, which expands to a lambda. (A `(delay e)` in Racket is similar to $\lambda 0. e$ in λ_{ZFC} , but it caches the value of `e` when it is first computed, and is applied using `force`.)

To test termination guarantees, we exhibit a few programs with different termination conditions. The first program never terminates:

```
(define/drabayes never-terminate (loop))
```

When asked for any number of samples in the preimage of any nonempty set, Dr. Bayes simply returns no samples:

```
> (drabayes-sample never-terminate 100 univ-set)
'(())
```

How long sampling takes depends on the probability with which \perp is chosen for branches: a lower probability of \perp increases the average number of loops before \perp is chosen, resulting in longer wait times. Any probability of \perp below $\frac{1}{3}$ is reasonable, and we generally use $\frac{1}{5}$.

The following program returns `0` with probability $\frac{1}{2}$, and otherwise loops forever:

```
(define/drabayes half-terminate
  (if (< (random) 1/2) 0 (loop)))
```

With the probability of \perp branches at $\frac{1}{5}$ and the probabilities of `true` and `false` at $\frac{2}{5}$, we should expect perhaps $\frac{2}{5}$ of the samples we ask for. However, we get almost all of them:

```
> (length (drabayes-sample half-terminate 500 univ-set))
486
```

This is due to the self-adjusting tree search. Each \perp branch choice in the fully inlined `(loop)` results in an empty preimage, so its corresponding leaf in the search tree is removed. The required adjustments reduce the probability the sampler chooses `false` for $(< (\text{random}) 1/2)$ in subsequent samples.

Roughly, the self-adjusting search “learns” that `(loop)` terminates with low probability and usually avoids it. The more samples are taken, the lower its probability estimate, though it never reaches zero.

With probability 1, the following program returns a geometrically distributed value.

```
(define/drabayes (geometric p)
  (if (< (random) p) 0 (+ 1 (geometric p)))))

(define/drabayes almost-surely-terminate
  (geometric 1/2))
```

Dr. Bayes rejects some samples:

```
> (length (drabayes-sample almost-surely-terminate 500 univ-set))
493
```

The rejected samples are from \perp branch choices. As with `half-terminate`, the self-adjusting search keeps the number of \perp choices small.

This program always terminates, but abstractly does not seem to:

```

(define/drabayes abstractly-loop
  (let ([x (random)]
        [y (random)])
    (if (< x y)
        (if (>= x y) (loop) 0)
        (if (< x y) (loop) 1))))

```

When $(< x y)$, $(\geq x y)$ is impossible, so the program returns 0. Otherwise, $(< x y)$ is impossible, so the program returns 1. Because `loop` is never applied, the program terminates.

However, the exact preimage of `true-set` under the interpretation of $(< x y)$ cannot be represented directly; its rectangular cover is `univ-omega-set`. The same is true for the exact preimage of `true-set` under $(\geq x y)$. In general, Dr. Bayes's implementation of `sample-part` can never rule out all branch choices that lead to `loop`.

Because `sample-part` may choose \perp for a branch choice, sampling terminates anyway, though for sequences of choices that contain \perp it rejects the sample. The proportion of samples accepted depends on how finely the program domain is partitioned. Dr. Bayes has a parameter `drabayes-max-splits` that determines how many times each projection is split in half: if set to n , the size of each projection's partition is 2^n . When set to 0, the partition whose cover is sampled from is that induced by the program's minimal branch traces T_* . With no splits, Dr. Bayes accepts about half the samples:

```

> (drabayes-max-splits 0)
> (length (drabayes-sample abstractly-loop 500 univ-set))
235

```

When set to a larger number, however, splitting the program domain allows preimage refinement to often determine that $(< x y)$ implies not $(\geq x y)$:

```

> (drabayes-max-splits 5)
> (length (drabayes-sample abstractly-loop 500 univ-set))
480

```

The only rectangular part covers that allow a later \perp branch choice are those that contain ω for which $(< x y)$ is true and others for which it is false; i.e. they straddle the line $\omega j_x = \omega j_y$ where j_x and j_y are the subcomputation indexes of the interpretations of each `(random)`.

Most of the examples so far require the possibility of \perp branch choices to terminate. However, most programs, like `almost-surely-terminate`, abstractly terminate with probability 1, and thus do not require it. Therefore, we define a parameter `drbayes-always-terminate?` and implement `sample-part` so that it chooses \perp only when `drbayes-always-terminate?` is set to `#t`. For example,

```
> (drbayes-always-terminate? #f)
> (length (drbayes-sample almost-surely-terminate 500 univ-set))
500
```

The default value is `#f`.

9.2 Primitives

The last chapter demonstrates sampling in a preimage under subtraction (Figure 8.17). Addition is no more difficult. In fact, by Theorem 8.9, once we have addition or subtraction it is easy to define the other so that it is just as efficient, because each is an axial inverse of the other. Alternatively, we could define one from the other using an additional negation primitive and $a - b = a + (-b)$ or $a + b = a - (-b)$.

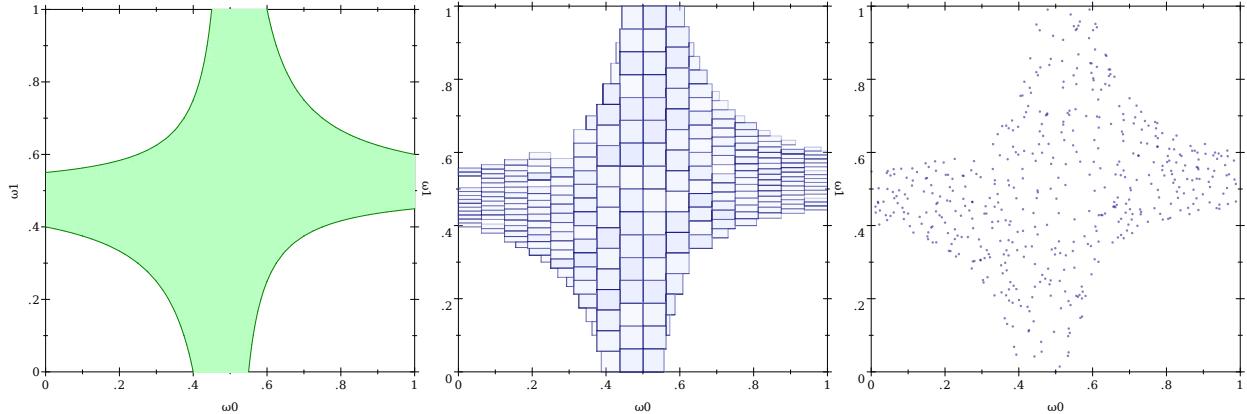
To help ensure Dr. Bayes is useful, it additionally has comparison primitives, multiplication and division primitives (for which one is easily defined in terms of the other), and various $\mathbb{R} \rightarrow \mathbb{R}$ function primitives such as inverse cumulative distribution functions (CDFs) for a few common distributions.

When subtraction and predicates are defined, comparing real numbers is easy. For example, if `negative?` is defined as in (8.35), then because $a < b$ if and only if $a - b < 0$,

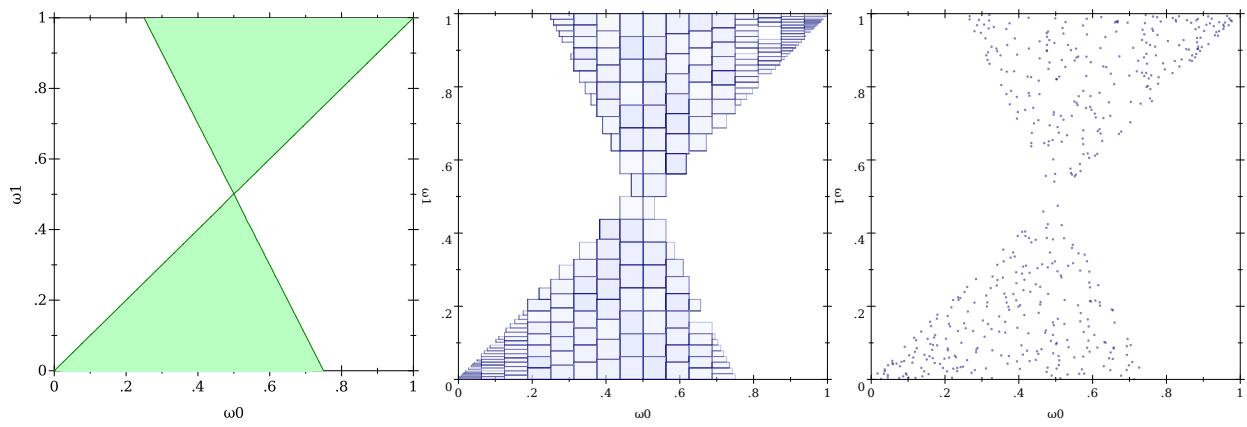
```
(define/drabayes (< a b)
  (negative? (- a b)))
```

The other comparison operators are defined similarly. We could define real equality by

```
(define/drabayes (= a b)
  (and (<= a b) (<= b a)))
```



(a) Preimage of $[-0.1, 0.2]$ under the interpretation of $(\text{uniform } -1 \ 1) \cdot (\text{uniform } -1 \ 1)$



(b) Preimage of $[-0.5, 1]$ under the interpretation of $(\text{uniform } -1 \ 1) / (\text{uniform } -1 \ 1)$

Figure 9.1: Exact preimages under multiplication and division, sampled part covers, and sampled points.

However, real equality is rarely useful because $(= \ a \ b)$ is usually a zero-probability event; i.e. the preimage of $\{\text{true}\}$ under the interpretation of $(= \ a \ b)$ usually has zero measure.

Figure 9.1a shows the result of sampling in the preimage of $[-0.1, 0.2]$ under the interpretation of either of the following equivalent programs:

<pre>(define/drabayes uniform-mul (* (uniform -1 1) (uniform -1 1)))</pre>	<pre>(define/drabayes uniform-mul (* (+ -1 (* 2 (random))) (+ -1 (* 2 (random)))))</pre>
--	--

Similarly, Figure 9.1b shows the result of sampling in the preimage of $[-0.5, 1]$ under the interpretation of $(/ \ (\text{uniform } -1 \ 1) \ (\text{uniform } -1 \ 1))$.

There are two points of interest here. The first is that the point samples in Figure 9.1 exhibit nonuniformity. (It is in the horizontal arms in the multiplication preimage, and in the

corners in the division preimage.) Nonuniformity arises from the fact that the probabilities with which part covers are chosen can only approximate the covered parts' true measures (Figure 8.16), and from sampling dependent uniform random variables in `sample-source*` (8.50). It is adjusted for by weighting the samples, and is further mitigated by the extension to the self-adjusting search that causes the search tree to converge to a tree with stated leaf probabilities (Section 8.4.5). But nonuniformity can lower the samples' information content in a query-dependent way. We quantify it further on.

The second point of interest is that `(uniform -1 1)` is defined by a function that encodes the **uniform distribution family**:

```
(define/drabayes (uniform a b)
  (+ a (* (- b a) (random))))
```

This function transforms uniform random variables on the unit interval into uniform random variables between `a` and `b`. The uniform distribution family is one of the simplest common examples of a **location-scale family**, which can always be encoded in Dr. Bayes as

```
(define/drabayes (family-name loc scale)
  (+ loc (* scale (standard-inv-cdf (random)))))
```

For the uniform distribution family, `standard-inv-cdf` is the identity function.

Another location-scale family, the normal distribution family, is encoded in Dr. Bayes using the implementation of `normal-inv-cdfpre` (8.32):

```
(define/drabayes (normal mu sigma)
  (+ mu (* sigma (normal-inv-cdf (random)))))
```

Other implemented families include the exponential, Cauchy, and logistic distribution families.

Some distribution families are significantly more difficult to encode. For example, gamma distributions are parameterized on a *shape* and a scale. Because one parameter is a scale, we can reduce the work to implementing a two-argument primitive `gamma-inv-cdf`:

```
(define/drabayes (gamma k θ)
  (* θ (gamma-inv-cdf k (random))))
```

Implementing `gamma-inv-cdf` requires implementing the following trijection, extended to a compact superdomain.

$$\begin{aligned} F_p : (0, +\infty) \times (0, +\infty) &\rightarrow (0, 1) && \text{Gamma CDF} \\ F_x : (0, +\infty) \times (0, 1) &\rightarrow (0, +\infty) && \text{Gamma inverse CDF} \\ F_k : (0, 1) \times (0, +\infty) &\rightarrow (0, +\infty) && ??? \end{aligned} \tag{9.1}$$

F_p and F_x are well-known, and Racket's `math/distributions` library exports quite accurate implementations of them. Unfortunately, as far as we can tell, no one else has ever needed an implementation of F_k . While the numerical analysis required to implement it accurately on its entire domain is beyond the scope of this work, we plan to do it in the future.

As soon as we have `gamma`, we can encode other distributions important in Bayesian modeling by encoding sampling algorithms for them that are defined in terms of gamma distributions. For example, the inverse gamma and beta families can be encoded by

```
(define/drabayes (inv-gamma k θ)
  (/ 1 (gamma k θ)))

(define/drabayes (beta α β)
  (let ([x (gamma α 1)]
        [y (gamma β 1)])
    (/ x (+ x y))))
```

The Dirichlet distribution family encoding would be similar to `beta`, but would use structural recursion over a list of parameters to get and normalize an unbounded number of gamma-distributed random variables.

In general, as with `beta`, the encoding of any sampling algorithm is also an encoding of the distribution it samples from. For example, this encoding of the Box-Muller algorithm [XXX: cite] also encodes the standard normal distribution:

```
(define/drabayes (normal/box-muller)
  (* (sqrt (* -2 (log (random)))))
    (partial-cos (* pi (uniform -1 1)))))
```

Here, `partial-cos` is the cosine function defined on $[-\pi, \pi]$, which is implemented by pasting together two symmetric, monotone pieces defined on $[-\pi, 0]$ and $[0, \pi]$.

Using `(normal-inv-cdf (random))` instead of `(normal/box-muller)` is much faster, consisting of one Ω projection and one primitive operation instead of two Ω projections and nine primitive operations. However, the point of defining `(normal/box-muller)` is to demonstrate the expressive power of a probabilistic language defined measure-theoretically.

A language based on probability density functions necessarily distinguishes between primitive and derived random variables. The former are simply projections, and conditions are restricted to be primitive random variable equalities so that conditional densities always exist. Conditions referring to a derived random variable such as `(normal/box-muller)`, or even `(normal μ σ)` using the present definition of `normal`, are close to unthinkable.

We will explore how to leverage this new expressiveness after showing how Bayesian theories with density models are encoded in Dr. Bayes, and showing how to use its sampling algorithm to answer conditional queries about them.

9.3 Theories With Density Models

9.3.1 Normal-Normal

We start with one of the simplest Bayesian theories, the normal-normal, introduced in Chapter 2. Specified constructively, it is

$$\begin{aligned} X &\sim \text{Normal}(0, 1) \\ Y &\sim \text{Normal}(X, 1) \end{aligned} \tag{9.2}$$

There are many possible encodings in Dr. Bayes whose interpretations are a model of this theory. One of the most straightforward is

```
(define/drabayes normal-normal
  (let* ([x (normal 0 1)]
         [y (normal x 1)])
    (cons x y)))
```

where `let*` makes `x` visible in the definition of `y` by expanding to nested `let` expressions, and `cons` constructs pairs.

In Chapter 2, we used Bayes' law for densities to find the distribution of X given $Y = 2$. By not defining the language in terms of densities, we have given up the ability to handle such zero-probability conditions except as limits, and we cannot wait for limits to complete. We will show that not having zero-probability conditions matters little.

But first, from a philosophical standpoint, the condition $Y = 2$ is hard to support: it is an assertion about *all of the countably many digits* of Y . If finding the distribution of $X | Y = 2$ is a typical inference task, this amounts to having infinite knowledge about a real-world observation. Still, zero-probability conditions are often convenient, so we intend to investigate supporting them in future work.

To sample within a positive-probability preimage, the condition needs to be wider; e.g. $Y \in [2 - \epsilon, 2 + \epsilon]$ where $\epsilon > 0$ is small. For `normal-normal`, we must sample within the preimage of $\mathbb{R} \times [2 - \epsilon, 2 + \epsilon]$. Equivalently, we could encode the condition in the program as a proposition about `y`:

```
(define/drabayes normal-normal/cond
  (let* ([x (normal 0 1)]
         [y (normal x 1)])
    (cons x (<= (- 2 epsilon) y (+ 2 epsilon)))))
```

and sample within the preimage of $\mathbb{R} \times \{\text{true}\}$.

Figure 9.2 shows the results of sampling 10000 times within the preimage for $\epsilon = 0.2$ and $\epsilon = 0.01$, each of which takes about 3 seconds on current hardware. The last plot is a density estimate of the distribution of $X | Y \in [1.8, 2.2]$, wherein $\epsilon = 0.2$, not 0.01. Even so, the density estimate is very good (and is consistently so in multiple tests), suggesting that

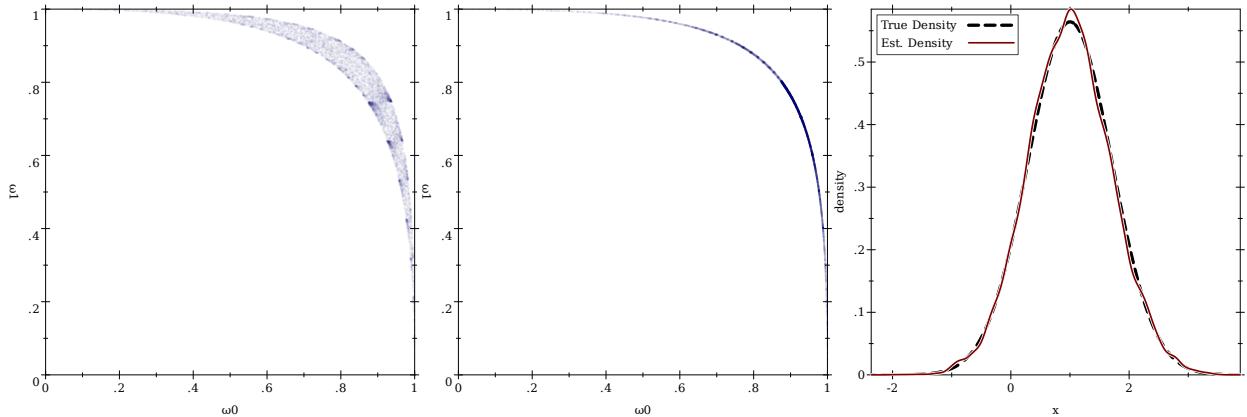


Figure 9.2: From left to right, samples from the preimage of $[1.8, 2.2]$, samples from the preimage of $[1.99, 2.01]$, and a density estimate computed from the image of the first set of samples.

we do not need anything close to a zero-probability condition. A fairly wide interval centered on 2 is enough.

For $\epsilon = 0.2$, sufficient statistics for the computed distribution are

```
> (mean xs ws)
0.990829840759573

> (stddev xs ws)
0.7026140368998269
```

where `xs` are the first of every sampled `(cons x y)`, and `ws` are the sample weights. The exact values of these statistics for $X \mid Y = 2$ are 1 and $\sqrt{\frac{1}{2}} \approx 0.707107$.

Because we are sampling, the value `(mean xs ws)` is itself a random variable. The standard way to quantify its uncertainty or information content is **Monte Carlo variance** [XXX: cite?]. To get Monte Carlo variance in the same units as the mean, we use `monte-carlo-stddev` from Racket's `math/statistics` library, which returns its square root:

```
> (monte-carlo-stddev xs ws)
0.007026140368998269
```

By the Central Limit theorem, the distribution of `(mean xs ws)` should be approximately normal. We can therefore fit a normal distribution to it to obtain a confidence interval.¹

¹There are ways more in line with Bayesian philosophy to quantify uncertainty about expected values computed using Monte Carlo methods, but they are more complicated than we need.

```

> (real-dist-hpd-interval
  (normal-dist (mean xs ws) (monte-carlo-stddev xs ws))
  0.95)
0.9770588586860135
1.0046008228331327

```

Here, `real-dist-hpd-interval` finds the High Probability Density (HPD) interval: the narrowest interval containing 95% of the area under the density of the distribution object returned by `normal-dist`. Thus, we are 95% confident that the mean of $X | Y = 2$ is between 0.977 and 1.004.

In this example, weighted samples seem to carry about as much information as unweighted. For comparison, here are the same results computed from 10000 unweighted samples chosen according to the exact distribution of $X | Y = 2$:

```

> (define zs (sample (normal-dist 1 (sqrt 1/2)) 10000))

> (mean zs)
0.9911222027798829

> (stddev zs)
0.695842224805195

> (monte-carlo-stddev zs)
0.006958422248051949

> (real-dist-hpd-interval
  (normal-dist (mean zs) (monte-carlo-stddev zs))
  0.95)
0.9774839457844788
1.004760459775287

```

Monte Carlo variance is a little lower, so the resulting confidence interval is a little tighter.

We can also compute probabilities as expected values, as discussed in Chapter 5. For example, $\Pr[X \in (0, 1) | Y = 2]$ is approximately

```

> (mean (map (indicator (λ (x) (< 0 x 1))) xs) ws)
0.42925920262068257

```

where `indicator` converts $X \Rightarrow \text{Bool}$ functions into $X \Rightarrow \{0, 1\}$ functions. As with the mean, this computed probability is also a random variable, but fitting a normal distribution using

Monte Carlo variance may be a bad idea: the fitted distribution would give positive probability to sets containing negative “probabilities.” It is better to fit a beta distribution, which has support only in $[0, 1]$, and is well-suited for characterizing distributions over probabilities.²

The `math/statistics` export `monte-carlo-prob-dist` does this for us:

```
> (real-dist-hpd-interval
  (monte-carlo-prob-dist (λ (x) (< 0 x 1)) xs ws)
  0.95)
0.4195626419077031
0.43896369932251544
```

Thus, we are 95% confident that $\Pr[X \in (0, 1) | Y = 2]$ is between about 0.419 and 0.439.

The following encoding of the same normal-normal theory uses the standard normal distribution as defined using the Box-Muller algorithm.

```
(define/drabayes normal-normal/box-muller
  (let* ([x (normal/box-muller)]
         [y (+ x (normal/box-muller))])
    (cons x y)))
```

The results (elided) from taking 10000 samples in the preimage of $[1.8, 2.2]$ are nearly identical, except the Monte Carlo standard deviation is a little higher, at approximately 0.00726 instead of 0.00703, and collecting the samples takes about 22 seconds instead of 3.

9.3.2 Normal-Normals

Extending the normal-normal theory with more observations requires adding more random variables that depend on X . A template for encoding them is

```
(define/drabayes normal-normals
  (let ([x (normal μ σ)])
    (list x (normal x σ₁) ... (normal x σₙ))))
```

²It is also easy to put this in line with Bayesian philosophy and get true credible intervals, by simply adding prior pseudocounts to the beta parameters.

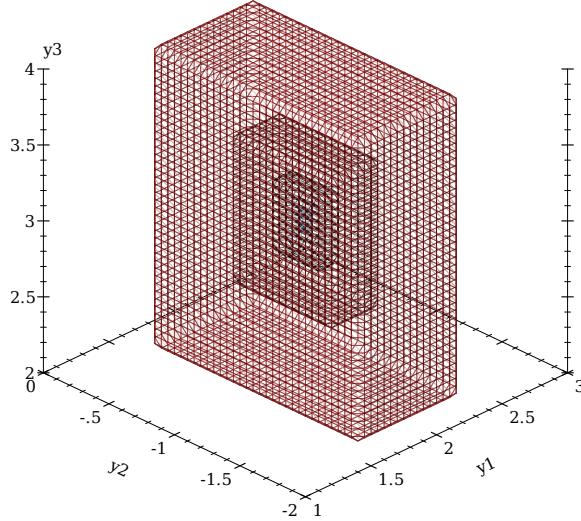


Figure 9.3: Nested rectangular conditions: each axis represents an observation $Y_i \in [y_i - \epsilon_i, y_i + \epsilon_i]$.

We sample within preimages of $\mathbb{R} \times [y_1 - \epsilon_1, y_1 + \epsilon_1] \times \dots \times [y_n - \epsilon_n, y_n + \epsilon_n] \times \{\langle \rangle\}$, where y_1, \dots, y_n are the observed data. (In functional languages, a finite list consists of nested pairs terminated by $\langle \rangle$.) Each observation y_i has its own interval width ϵ_i .

Figure 9.3 illustrates nested condition sets $[y_1 - \epsilon_1, y_1 + \epsilon_1] \times [y_2 - \epsilon_2, y_2 + \epsilon_2] \times [y_3 - \epsilon_3, y_3 + \epsilon_3]$ as each ϵ_i decreases. If the condition sets converge to $\{\langle y_1, y_2, y_3 \rangle\}$, then queries conditioned on $Y_1 = y_1, Y_2 = y_2, Y_3 = y_3$ likewise converge.

It is natural to wonder what the relationships must be among $\epsilon_1, \dots, \epsilon_n$ to ensure convergence. By illustrating with nested rectangles, not cubes, Figure 9.3 suggests they need not be equal. Must they have the same order of magnitude, be proportional, or at least be functions of each other? Not at all. A formal statement and proof are in Appendix B.

This fact suggests another that is possibly counterintuitive, but fortunate: that a wide observation $Y_1 \in [y_1 - 0.2, y_1 + 0.2]$ may affect the distribution of X as much as a narrow observation $Y_2 \in [y_2 - 0.01, y_2 + 0.01]$. Figure 9.4 demonstrates precisely this using the `normal-normals` template with two observations $y_1 := 2$ and $y_2 := -1$, all standard deviations set to 1, and two different settings for ϵ_1, ϵ_2 . The left plot shows 10000 samples from the preimage using $\epsilon_1 = 0.2, \epsilon_2 = 0.01$. The middle plot shows 10000 samples using

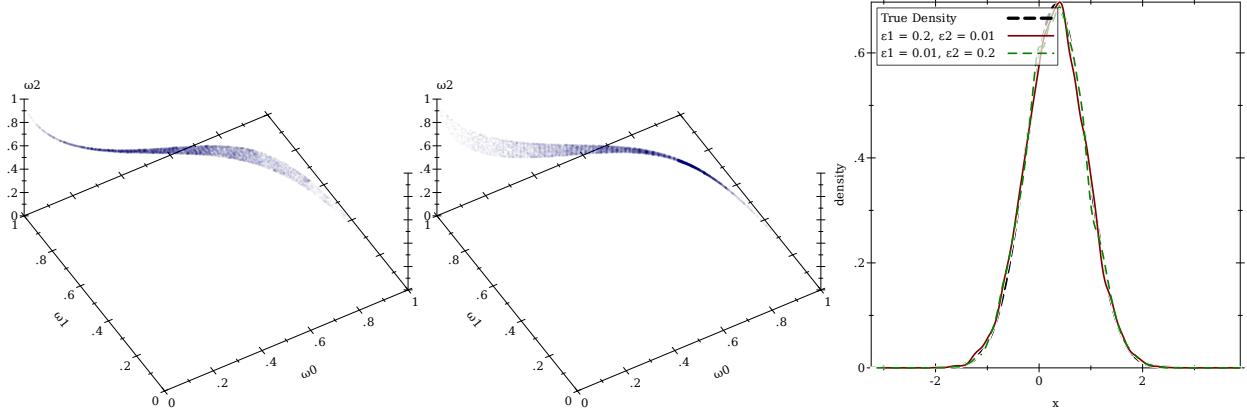


Figure 9.4: From left to right, samples from the preimage of a condition using $\epsilon_1 = 0.2, \epsilon_2 = 0.01$, samples using $\epsilon_1 = 0.01, \epsilon_2 = 0.2$, and density estimates computed from the images of the sets of samples.

$\epsilon_1 = 0.01, \epsilon_2 = 0.2$. Though the preimage sets are visibly different, X 's estimated density is nearly the same either way: close to the true distribution, a normal with $\mu = \frac{1}{3}$ and $\sigma = \sqrt{\frac{1}{3}}$. In both cases, the μ estimate's Monte Carlo standard deviation is about 0.00575.

9.3.3 Polynomial Fitting

When encoding and computing conditional queries about normal-normal theories, it is easy to notice that the time it takes Dr. Bayes to sample is superlinear in the number of observations. In fact, this is expected. Because Ω subsets are represented by trees that correspond with the expression tree and projections are looked up and updated starting from the root, a `(random)` expression's image and preimage computation takes time proportional to its depth in the fully inlined program. Because `(list e1 e2 ... en)` is equivalent to `(cons e1 (cons e2 (cons ... (cons en null))))`, if each e_i has one `(random)` subexpression at a constant depth, its overall time complexity is $O(n^2)$.

We are fine with this for now. At this early stage, theoretical simplicity is more important than speed. Additionally, it gives us a fresh problem on which to demonstrate using Dr. Bayes for Bayesian regression, particularly to infer the distribution of a random function from number of observations to running time.

Figure 9.5 demonstrates using Dr. Bayes to reason about the behavior of Dr. Bayes.

```

(define xs  (list 0 1 2 3 4 5 6 7 ...))
(define ys* (list 0.063 0.262 0.493 0.814 1.222 1.708 2.238 2.883 ...))

(define/drabayes (quadratic-eval a0 a1 a2 x)
  (+ a0 (* x (+ a1 (* x a2)))))

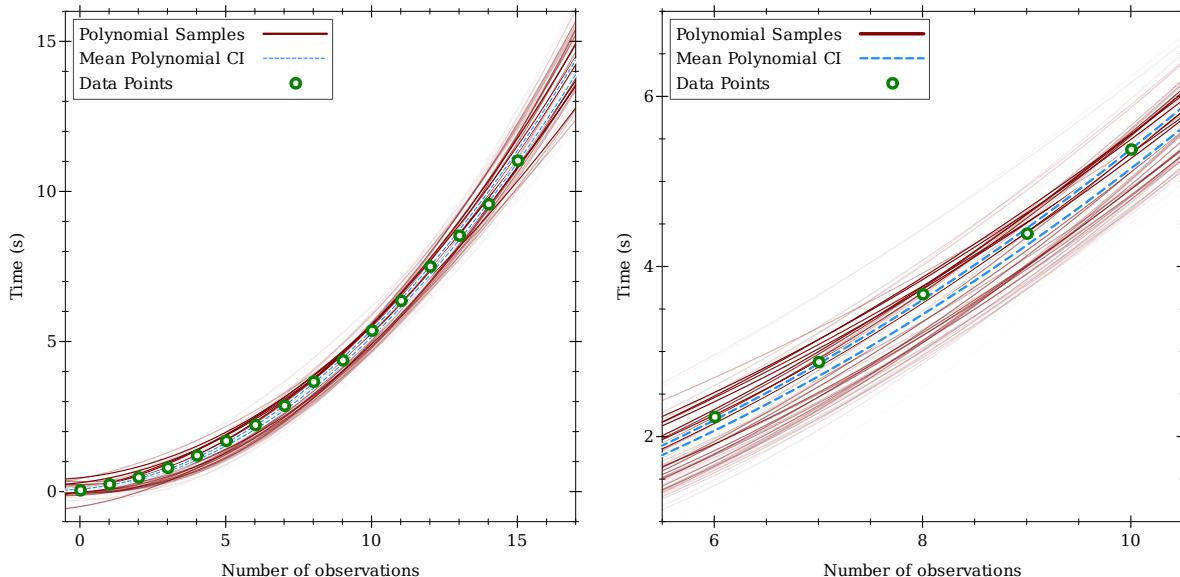
(define/drabayes (generate a0 a1 a2 xs)
  (if (null? xs) null (cons (normal (quadratic-eval a0 a1 a2 (first xs)) 1)
                             (generate a0 a1 a2 (rest xs)))))

(define/drabayes (condition ys ys*)
  (if (null? ys) #t (and (let ([y (first ys)])
                            [y* (first ys*)])
                           (<= (- y* 0.001) y (+ y* 0.001)))
      (condition (rest ys) (rest ys*))))

(define/drabayes normal-normal-running-time
  (let* ([a0 (cauchy 0 0.1)]
         [a1 (cauchy 0 0.1)]
         [a2 (cauchy 0 0.1)]
         [ys (generate a0 a1 a2 xs)])
    (list a0 a1 a2 (condition ys ys*))))

```

(a) A Bayesian theory of the running time of Dr. Bayes as a quadratic function of the number of observations.



(b) The observations, sampled polynomials given observations, and a 95% confidence interval for the average polynomial.

Figure 9.5: Bayesian analysis of Dr. Bayes's running time, using Dr. Bayes.

The theory is as follows: let a_0, a_1, a_2 be random coefficients that define a quadratic function $f(x) := a_0 + a_1 \cdot x + a_2 \cdot x^2$, where x is the number of observations. We assume that the time to take 1000 samples conditioned on x observations is $f(x)$ seconds plus some normally distributed noise. We are interested in the distribution of f (equivalently the distribution of $\langle a_0, a_1, a_2 \rangle$) given some running time observations.

A priori, we do not know much about a_0, a_1 and a_2 , so we assume they have Cauchy distributions, which are bell-shaped like normal distributions but allow much more variation. We also allow the random noise added to $f(x)$ to be fairly large (standard deviation 1 second).

We collected running times for 0 through 15 observations with millisecond accuracy. This data is encoded as `xs` (numbers of observations) and `ys*` (observed times) in Figure 9.5a. The function `quadratic-eval` evaluates f . The `generate` function returns a list of running time random variables generated from `xs`. The `condition` function returns a boolean random variable which is `#t` only when every `y` in `ys` is near its corresponding observation `y*` in `ys*`. Because our times have millisecond accuracy, `condition` requires each `y` to be within a millisecond of `y*`, or it returns `#f`.

The encoding of our theory outputs a list containing a_0, a_1, a_2 and the value of the condition. Sampling within the preimage of $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \{\text{true}\} \times \{\langle \rangle\}$, and taking the image of these samples, results in samples from the distribution of f 's coefficients given the observations.

Figure 9.5b shows two views of a plot of the observations, the polynomial function samples, and an inferred 95% confidence interval computed by fitting a normal distribution to the computed mean of each coefficient and its Monte Carlo standard deviation. By our prior knowledge and the close fit, we believe the quadratic model is a good one.

In fact, we will use it to make a prediction. By evaluating the sampled polynomials on the value 50, we get samples from a distribution over running times with mean about 114 and standard deviation about 13.7. Its upper and lower 2.5% quantiles are approximately 89 and 141; thus, if the model is correct, then given the observations, there is a 95% probability

that the running time for 50 normal-normal observations is between approximately 89 and 141 seconds.

9.3.4 Model Selection

When Bayesian practitioners have two competing theories in mind to explain some phenomenon, they perform **model selection** to determine which is most probable. Of course, we call it ***theory selection***.

As a constructive theory, it is

$$M \sim [m_1 \mapsto p_1, m_2 \mapsto p_2]$$

$$\Theta \sim \begin{cases} \text{Prior}_1 & \text{if } M = m_1 \\ \text{Prior}_2 & \text{if } M = m_2 \end{cases} \quad Y \sim \begin{cases} \text{Likelihood}_1(\Theta) & \text{if } M = m_1 \\ \text{Likelihood}_2(\Theta) & \text{if } M = m_2 \end{cases} \quad (9.3)$$

where p_1 and p_2 are the probabilities of theories m_1 and m_2 , and Θ is a random vector of parameters for one theory or the other. A concrete, though contrived example is

$$M \sim [cc \mapsto \frac{1}{2}, nn \mapsto \frac{1}{2}]$$

$$X \sim \begin{cases} \text{Cauchy}(0, 1) & \text{if } M = cc \\ \text{Normal}(0, 1) & \text{if } M = nn \end{cases} \quad Y \sim \begin{cases} \text{Cauchy}(X, 1) & \text{if } M = cc \\ \text{Normal}(X, 1) & \text{if } M = nn \end{cases} \quad (9.4)$$

Here, the competing theories are Cauchy-Cauchy and normal-normal.

The major task in theory selection is computing the distribution of $M | Y = y$, to determine the probabilities of each theory given observed data. Theoretically, it is as simple as applying a version of Bayes' law for mixed masses and densities:³ if $f_Y(y) > 0$, then

$$p_{M|Y}(m | y) = \frac{p_M(m) \cdot f_{Y|M}(y | m)}{f_Y(y)}$$

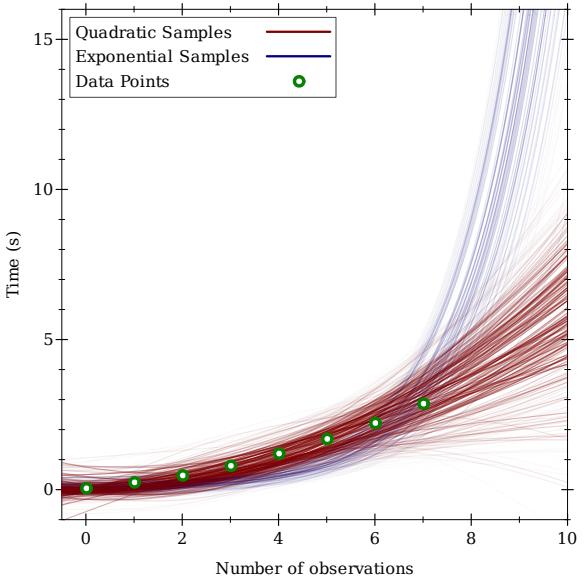
$$= \frac{p_M(m) \cdot f_{Y|M}(y | m)}{\sum_{m \in \{m_1, m_2\}} p_M(m) \cdot f_{Y|M}(y | m)} \quad (9.5)$$

³Technically, every version of Bayes' law that Bayesians use is Bayes' law for Radon-Nikodým derivatives.

```
(define/drbayes quad-or-exp-running-time
  (if (< (random) 1/2)
      ;; Quadratic running time
      (let* ([a0 (cauchy 0 0.1)]
             [a1 (cauchy 0 0.1)]
             [a2 (cauchy 0 0.1)]
             [ys (generate-quad a0 a1 a2 xs)])
        (list #t (list a0 a1 a2)
              (condition ys ys*)))
      ;; Exponential running time
      (let* ([b0 (cauchy 0 0.1)]
             [b1 (exponential 0.5)]
             [ys (generate-exp b0 b1 xs)])
        (list #f (list b0 b1)
              (condition ys ys*)))))

```

(a) Encoding of Bayesian theory selection



(b) Data points and sampled running time functions

Figure 9.6: Bayesian theory selection in Dr. Bayes.

Unlike using Bayes' law for densities in Chapter 2, this is *not* conveniently in terms of functions we have on-hand. While we have $p_M = [m_1 \mapsto p_1, m_2 \mapsto p_2]$, we do not have the conditional density $f_{Y|M}$. Much of the literature on Bayesian theory selection is devoted to efficiently computing $f_{Y|M}$ or approximations of it that can be used in certain circumstances.

Sometimes, the combined theory has a density model that is amenable to traditional Monte Carlo methods. (An example is our contrived Cauchy-Cauchy vs. normal-normal theory.) Practitioners need only sample according to the density conditioned on the data, and count the frequencies with which $M = m_1$ and $M = m_2$.

In Dr. Bayes, that always works. For example, suppose we want to determine whether Dr. Bayes's normal-normal running time is best modeled by a distribution over quadratic functions $f(x) := a_0 + a_1 \cdot x + a_2 \cdot x^2$ or exponential functions $g(x) := b_0 + b_1 \cdot 2^x$. As with a_0 , we assume b_0 has a Cauchy distribution. But b_1 should not be negative nor too large, so instead of a Cauchy distribution, we assume it has an exponential distribution with scale $\frac{1}{2}$. A priori, we assume the quadratic and exponential theories are equally likely.

Figure 9.6 shows some of the encoding of the combined theory. (The rest is similar to

`quadratic-eval` and `generate` in Figure 9.5a.) It also shows the results of sampling in the conditioned model. We use only 8 observations, as adding more makes the exponential theory too unlikely; for example, a typical run with 16 observations returns no exponential function samples. With 8 observations, we compute

```
> (real-dist-hpd-interval
  (monte-carlo-prob-dist (λ (m) m) ms ws)
  0.95)
0.8543205426853948
0.8858619943029161
```

where `ms` is the list of boolean theory choices, in which `#t` represents choosing quadratic. Thus, we are 95% confident that the probability of the quadratic theory, given the 8 observations, is between 0.85 and 0.89. Given 16 observations, the estimated probability of the quadratic theory is a solid 1.

9.4 Theories Without Density Models

While Dr. Bayes has not yet constructed a density model for any of our theories, all of them so far have density models. (Even the theory selection theory does, though its density model is not particularly amenable to Monte Carlo methods.) In this section, we demonstrate some useful theories that do not have density models.

9.4.1 Bounded Measuring Devices

The simplest theories without density models try to faithfully model measuring devices, which in reality cannot output unbounded values.

Suppose we wish to model a thermometer that would report the actual temperature with normally distributed noise, except that it cannot display numbers less than 0 or greater than 100. Here is a theory in which we assume the true temperature is F :

$$\begin{aligned} T' &\sim \text{Normal}(F, 1) \\ T &= \max(0, \min(T', 100)) \end{aligned} \tag{9.6}$$

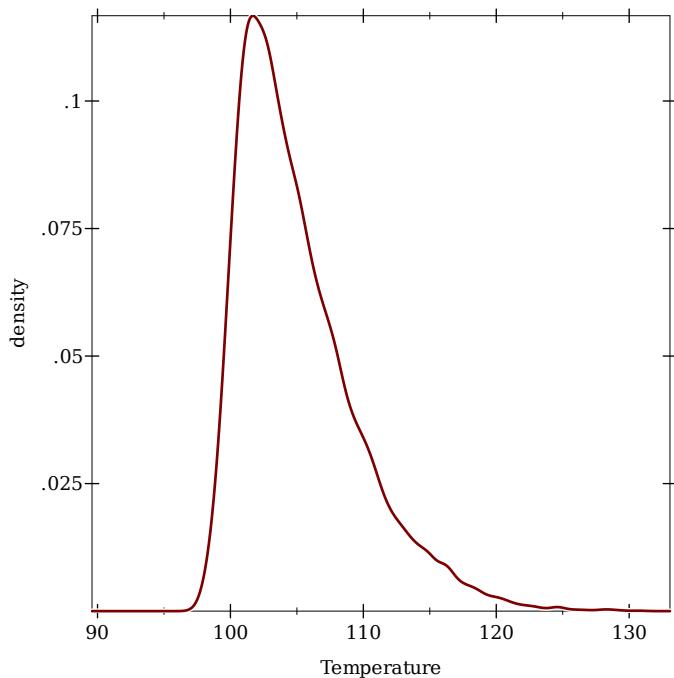


Figure 9.7: Distribution over temperature given that our thermometer displays its maximum value 100.

A typically Bayesian task would be to infer the distribution of the outside temperature given a thermometer reading. An encoding in Dr. Bayes is

```
(define/drabayes temp-outside
  (let* ([temp   (normal 90 10)]
         [therm  (max 0 (min 100 (normal temp 1))))]
    (cons temp therm)))
```

where `(normal 90 10)` represents our prior knowledge about the outside temperature.

Suppose we see the thermometer pegged at 100. What is the distribution of the outside temperature?

Figure 9.7 plots a density estimate of the image of samples taken in the preimage of $\mathbb{R} \times [100, 100]$. The distribution is skewed left, as we might expect: its mode is approximately 102, and the temperatures below 102 that can cause the thermometer to read 100 have lower probability than the temperatures above 102 that can cause it. The distribution is wide, corresponding with the fact that reading the maximum value does not tell us much about the probability of temperatures above the maximum value.

With 95% probability, the outside temperature is between about 98.4 and 114.5:

```
> (real-hpd-interval 0.95 temps ws)
98.47185450009886
114.45483575358433
```

As the result of a random simulation, these are of course random variables. But they are not expected values, so we cannot use Monte Carlo variance to quantify uncertainty in them.⁴ Still, the mean's Monte Carlo standard deviation is about 0.045, which indicates that they should be quite close to the true HPD interval.

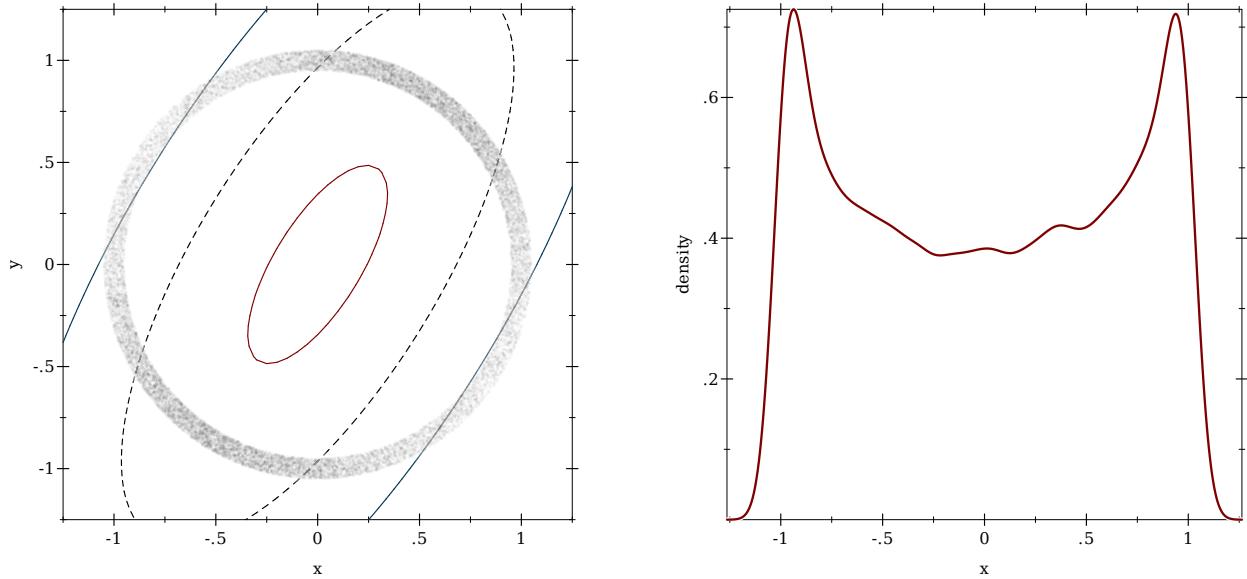
While it may seem useless to infer that we know very little, it is in these cases that Bayesian inference shines. In low-knowledge situations, prior knowledge becomes more important. Bayesian theories explicitly represent prior knowledge, which allows inference to fill in the gaps. See, for instance, Toronto et al [XXX: cite], wherein defaced portions of a photograph are modeled by missing data (i.e. no knowledge). A prior distribution that weakly favors continuous edges and contiguous color regions allows inference to fill in the missing data quite accurately. This work in particular, which tries to faithfully model the process of taking a photograph, could have benefited from not requiring a density model, to account for the fact that a camera is a bounded measuring device.

9.4.2 Non-Axial Conditions

In general, asserting a non-axial, zero-probability condition on a theory with a density model results in a theory without a density model. For example, suppose we augment our original normal-normal theory encoding with the distance from the origin:

```
(define/drabayes normal-normal/distance
  (let* ([x (normal 0 1)]
         [y (normal x 1)])
    (list x y (sqrt (+ (sqr x) (sqr y))))))
```

⁴Quantifying uncertainty about HPD intervals requires more complicated methods than we have discussed, such as bootstrapping.



(a) 42144 samples from the distribution of $X, Y | \sqrt{X^2 + Y^2} \in [1 - \epsilon, 1 + \epsilon]$, overlaid on a contour plot of the original density model

(b) Density estimate of the distribution of $X | \sqrt{X^2 + Y^2} \in [1 - \epsilon, 1 + \epsilon]$

Figure 9.8: Sampling within circular probabilistic conditions.

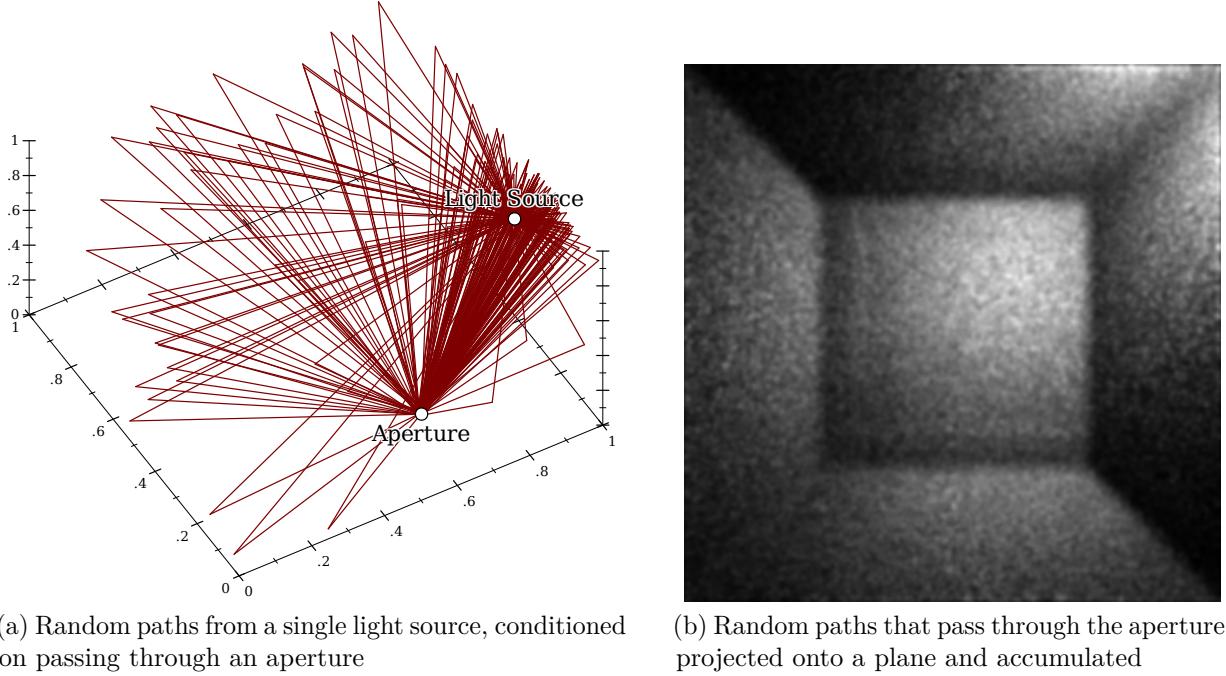
Figure 9.8 shows the results of sampling within the preimage of $\mathbb{R} \times \mathbb{R} \times [1 - \epsilon, 1 + \epsilon] \times \{\langle \rangle\}$, where $\epsilon = 0.05$. We are therefore conditioning on `x` and `y` being close to the unit circle.

We use $\epsilon = 0.05$ to make the left plot's circular band of samples wide, so it is obvious that the point densities correspond with the density model for [normal-normal/distance](#). Dr. Bayes samples just as efficiently for any $\epsilon > 0$. Therefore, while in the limit as ϵ approaches zero there is no density model, Dr. Bayes can still compute converging sequences of answers to queries.

We requested 50000 samples and received 42144. The rejected samples are from sampling within overapproximations, which sometimes causes preimage refinement to return \emptyset . We illustrate why further on.

9.4.3 Stochastic Ray Tracing

By implementing a small vector math library and some collision detection functions, we can encode a simple theory of light transport in Dr. Bayes for which conditional queries



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

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

Figure 9.9: Stochastic ray tracing in Dr. Bayes.

carry out stochastic ray tracing [54]. For example, the following functions compute the dot product between two vectors (represented by lists), and return a random vector with a uniform direction (for emitting light).

```
(define/drabayes (vec-dot v1 v2)
  (+ (* (list-ref v1 0) (list-ref v2 0))
      (* (list-ref v1 1) (list-ref v2 1))
      (* (list-ref v1 2) (list-ref v2 2)))))

(define/drabayes (uniform-vec)
  (list (normal) (normal) (normal)))
```

We also implement `vec-add`, `vec-neg` (negation) and `vec-scale` (elementwise multiplication by a constant). We additionally define a structure to represent collisions:

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

and a function to compute ray-plane intersections:

```

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

```

Figure 9.9a illustrates the main idea behind stochastic ray tracing. Using the vector functions, we simulate casting photons uniformly from a light source and reflect them uniformly when they collide with the walls of a square room, which generates paths. We condition the paths on passing through a small aperture, collect samples, and project them onto a collector plane on the other side of the aperture. Figure 9.9b shows the result of accumulating the collisions on the collector. The smaller the aperture, the smaller the probability a path passes through it, and the more focused the resulting image.

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

9.5 Current Shortcomings

To make progress, we must undertake the painful process of characterizing Dr. Bayes's shortcomings. We do not believe any of them are insurmountable, so we regard them as a guide for future work.

While Dr. Bayes is *efficient*, in the sense that the time complexity of sampling is a polynomial of low degree, it is not fast. In particular, the ray tracer takes around 10000 times the time a hand-coded MCMC sampler would take to render the same image. Much of the slowdown comes from computing images and preimages for deep (`random`) expressions, which is quadratic in the depth of the expression.

The ray-traced image in Figure 9.9b is comprised of 20 million samples. The image is only 256×256 , or 65536 pixels; it therefore could be much less grainy. The culprit is almost

certainly importance sampling, which is known to suffer from high Monte Carlo variance in high-dimensional spaces.

We could not set `drbayes-max-splits` to 5 before running the query that carried out stochastic ray tracing. The search tree became too large to fit in memory.

Dr. Bayes has no string or integer data types. Symbols can be faked with user-defined data structures, but direct support would be nice. We have done inference on probabilistic context-free grammars specified very naturally as mutually recursive functions. We did not demonstrate them here because, lacking symbols, the languages consist of lists of `#t` and `#f`, making them hard to understand.

Lambdas would have made the encoding of the theory of quadratic running time in Figure 9.5 shorter. The `generate` function is an easy application of `map`, and `condition` is an easy application of `andmap`—if we have lambdas.

With Dr. Bayes’s current primitives, we can find no good way to encode the gamma distribution family, beta distribution family and Dirichlet distribution family, all of which are important in Bayesian practice. Dr. Bayes needs a two-argument gamma primitive, from which these families can be defined.

The Bernoulli, multinomial, geometric, and other useful discrete distribution families can be encoded using `random`, `<`, and recursive functions, and conditions may contain the results of applying them. But defined this way, they are very slow.

There should be a way to specify arbitrary discrete distributions as mappings from values to probabilities.

Errors are often hard to find. Consider this bad program:

```
(define/drabayes bad  
  (+ 1 (list (normal 0 1))))
```

If we try to sample, we get this unhelpful error message:

```
drbayes-sample: cannot sample from the empty set
```

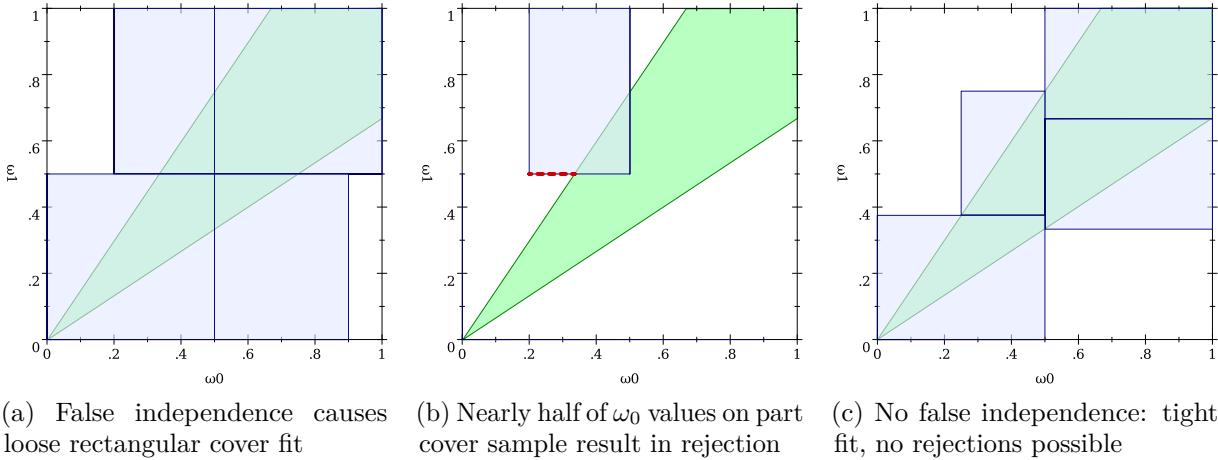


Figure 9.10: The dependency problem: falsely assuming independence of variables that occur more than once in an expression causes the rectangular preimage cover to fit loosely. If $\{\omega_0\}$ is drawn from the left of the part cover in (b), refinement returns \emptyset .

The culprit here is not distinguishing between errors and nontermination: they are both \perp in the semantics. In the exact semantics, computing preimages removes inputs that produce \perp , so errors do not occur. To begin addressing this, Dr. Bayes needs a separate type of error values, as well as a representation for rectangular sets of error values.

Our theory encodings returned every random variable of interest in a list so we could compute their expected values and study their behavior. Assembling theories from many such pieces could get quite verbose. It seems we need not just functions, but an additional form of abstraction that is more transparent by default.

Bayesians tend to think in terms of propositions, not sets. Encoding conditions as propositions, returning their boolean values, and sampling within the preimage of a rectangle with a `{true}` axis is a good start, but there should be a way to hide the details from users.

For the normal-normal theory encoding with the circular condition, we requested 50000 samples and received 42144. A much simpler program exhibits the same behavior:

```
(define/drabayes dependency-problem
  (let ([x (random)]
        [y (random)])
    (/ x (+ x y))))
```

Figure 9.10a shows the preimage of $[0.4, 0.6]$ and the rectangular cover Dr. Bayes samples from. If it samples the part cover in Figure 9.10b, it will reject a good portion of the ω samples because the part cover does not fit the preimage set as tightly as possible.

The reason is that x appears in $x/(x + y)$ twice. Suppose $x \in [0, 1]$ and $y \in [0, 1]$; then $x + y \in [0, 2]$. If we falsely assume each occurrence of x is independent, we conclude that $x/(x + y) \in [0, +\infty)$ because x/ϵ for any $\epsilon > 0$ is possible. But the actual range of $x/(x + y)$ is $[0, 1]$. In interval arithmetic, this is called the **dependency problem**.

Because images are computed compositionally and are rectangular, every occurrence of every variable is assumed independent. Computing preimages does the same.

We can mitigate the problem by partitioning more finely, which works well enough but takes extra time and space.

We can solve the problem occasionally by refactoring expressions. For example, if not $x = y = 0$, then $x/(x + y) = 1/(1 + y/x)$. In our program, $x = y = 0$ is a zero-probability event, so we can safely forget about it and rewrite the program as

```
(define/drbayes no-dependency-problem
  (let ([x (random)]
        [y (random)])
    (/ 1 (+ 1 (/ y x)))))
```

Figure 9.10c shows the rectangular cover Dr. Bayes samples from, which tightly fits the preimage.

9.6 Conclusions

Dr. Bayes is a proof-of-concept implementation, designed for exploring the expressive power and utility of the let-calculus whose semantics is defined in Chapter 7. We have shown that it is expressive enough to encode Bayesian theories with and without density models, including theories that (likely because they lack density models) are rarely regarded as Bayesian. Even at this very early stage, as a mostly direct implementation of the semantics with barely any

work put into making it fast, the implementation of the semantics is useful, at least for small problems.

We have demonstrated that allowing only positive-probability conditions matters little: they are philosophically easier to support, interval observations may be quite wide and still be as effectful as point observations, and interval observations' widths may be entirely independent.

We have characterized Dr. Bayes's shortcomings and plan to use them to drive future work.

Chapter 10

Related Work

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

10.1 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 [57, 58] have defined the semantics of *nonstandard interpretations* that enable efficient inference, but do not define the languages. [XXX: recheck, relate to computation indexing]

10.2 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 λ -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 λ -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.

10.3 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: $\llbracket \cdot \rrbracket_{\text{pre}^*}^\Downarrow$ is the concrete semantics, $\llbracket \cdot \rrbracket_{\text{pre}^*}^\Psi$ 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 λ_{ZFC} -computable combinators, not by a semantic function. This cleanly separates the syntax from the semantics, and allows us to prove the exact semantics correct mostly by proving simple categorical properties.

10.4 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 11

Conclusions and Future Work

11.1 Conclusions

We started by defining λ_{ZFC} , a call-by-value λ -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.

11.2 Future Work

Future work falls into two categories: expressiveness and optimization.

11.2.1 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 `{true}` under the test, and the *false* branch to the preimage of `{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.

11.2.2 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.

Appendix A

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.

A.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 σ -algebra of measurable sets.

Definition A.1 (σ -algebra, measurable set). *A collection of sets $\mathcal{A} \subseteq \mathcal{P}(X)$ is called a **σ -algebra** on X if it contains X and is closed under complements and countable unions. The sets in \mathcal{A} are called **measurable sets**.*

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

The analogue of continuity is measurability.

Definition A.2 (measurable mapping). *Let \mathcal{A} and \mathcal{B} be σ -algebras on X and Y . A mapping $g : X \rightarrow Y$ is **\mathcal{A} - \mathcal{B} -measurable** if for all $B \in \mathcal{B}$, preimage $g^{-1}(B) \in \mathcal{A}$.*

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

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

Product σ -algebras are defined analogously to product topologies.

Definition A.3 (finite product σ -algebra). *Let \mathcal{A}_1 and \mathcal{A}_2 be σ -algebras on X_1 and X_2 , and define $X := X_1 \times X_2$. The **product σ -algebra** $\mathcal{A}_1 \otimes \mathcal{A}_2$ is the smallest (i.e. coarsest) σ -algebra for which mapping $\text{fst } X$ and mapping $\text{snd } X$ are measurable.*

Definition A.4 (arbitrary product σ -algebra). *Let \mathcal{A} be a σ -algebra on X . The **product σ -algebra** $\mathcal{A}^{\otimes J}$ is the smallest σ -algebra for which, for all $j \in J$, mapping $(\pi_j : J \rightarrow X)$ is measurable.*

A.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 A.5 (measurable mapping arrow computation). *Let \mathcal{A} and \mathcal{B} be σ -algebras on X and Y . A computation $g : X \rightsquigarrow Y$ is **\mathcal{A} - \mathcal{B} -measurable** if $g|_{A^*}$ is an \mathcal{A} - \mathcal{B} -measurable mapping, where A^* is g 's maximal domain.*

Theorem A.6 (maximal domain measurability). *Let $g : X \rightsquigarrow Y$ be an \mathcal{A} - \mathcal{B} -measurable mapping arrow computation. Its maximal domain A^* is in \mathcal{A} .*

Proof. Because $g|A^*$ is measurable, preimage $(g|A^*)^{-1}(Y) = A^*$ is in \mathcal{A} . \square

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

Lemma A.7. *Let $g : X \rightarrow Y$ be an \mathcal{A} - \mathcal{B} -measurable mapping. For any $A \in \mathcal{A}$, restrict $g|A$ is \mathcal{A} - \mathcal{B} -measurable.*

Theorem A.8. *Let $g : X \rightsquigarrow_{\text{map}} Y$ be an \mathcal{A} - \mathcal{B} -measurable mapping arrow computation with maximal domain A^* . For all $A \subseteq A^*$ with $A \in \mathcal{A}$, $g|A$ is \mathcal{A} - \mathcal{B} -measurable.*

Proof. By Theorem 7.44 (mapping arrow restriction) and Lemma A.7. \square

We do not need to prove all interpretations using $\llbracket \cdot \rrbracket_a$ are measurable. However, we do need to prove mapping arrow combinators preserve measurability.

A.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 A.9 (images of preimages). *If $g : X \rightarrow Y$ and $B \subseteq Y$, $\text{image } g(\text{preimage } g^{-1}(B)) \subseteq B$.*

Lemma A.10 (expanded post-composition). *Let $g_1 : X \rightarrow Y$ and $g_2 : Y \rightarrow Z$ such that range $g_1 \subseteq \text{domain } g_2$, and let $g'_2 : Y \rightarrow Z$ such that $g_2 \subseteq g'_2$. Then $g_2 \circ_{\text{map}} g_1 = g'_2 \circ_{\text{map}} g_1$.*

Theorem A.11 (mapping arrow monotonicity). *Let $g : X \rightsquigarrow_{\text{map}} Y$. For any $A' \subseteq A \subseteq A^*$, $g|A' \subseteq g|A$.*

Proof. By Theorem 7.44 (mapping arrow restriction). \square

Theorem A.12 (maximal domain subsets). *Let $g : X \rightsquigarrow_{\text{map}} Y$. For all $A \subseteq A^*$, $\text{domain}(g|A) = A$.*

Proof. Follows from Theorem 7.45. \square

Now we can prove measurability.

Lemma A.13 ((\circ_{map}) measurability). *If $g_1 : X \rightarrow Y$ is \mathcal{A} - \mathcal{B} -measurable and $g_2 : Y \rightarrow Z$ is \mathcal{B} - \mathcal{C} -measurable, then $g_2 \circ_{\text{map}} g_1$ is \mathcal{A} - \mathcal{C} -measurable.*

Theorem A.14 ((\ggg_{map}) measurability). *If $g_1 : X \rightsquigarrow_{\text{map}} Y$ is \mathcal{A} - \mathcal{B} -measurable and $g_2 : Y \rightsquigarrow_{\text{map}} Z$ is \mathcal{B} - \mathcal{C} -measurable, then $g_1 \ggg_{\text{map}} g_2$ is \mathcal{A} - \mathcal{C} -measurable.*

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

$$(g_1 \ggg_{\text{map}} g_2) A^{**} = \text{let } g'_1 := g_1 A^{**} \\ g'_2 := g_2 (\text{range } g'_1) \\ \text{in } g'_2 \circ_{\text{map}} g'_1 \quad (A.1)$$

By Theorem A.8, g'_1 is an \mathcal{A} - \mathcal{B} -measurable mapping. Unfortunately, g'_2 may not be \mathcal{B} - \mathcal{C} -measurable when $\text{range } g'_1 \notin \mathcal{B}$.

Let $g''_2 := g_2 B^*$, which is a \mathcal{B} - \mathcal{C} -measurable mapping. By Lemma A.13, $g''_2 \circ_{\text{map}} g'_1$ is \mathcal{A} - \mathcal{C} -measurable. We need only show that $g'_2 \circ_{\text{map}} g'_1 = g''_2 \circ_{\text{map}} g'_1$, which by Lemma A.10 is true if $\text{range } g'_1 \subseteq \text{domain } g'_2$ and $g'_2 \subseteq g''_2$.

By Theorem A.12, $A^{**} \subseteq A^*$ implies $\text{domain } g'_1 = A^{**}$. By Theorem A.11 and Lemma A.9,

$$\begin{aligned} \text{range } g'_1 &= \text{image}(g_1 A^{**}) (\text{preimage}(g_1 A^*) B^*) \\ &= \text{image}(g_1 A^*) (\text{preimage}(g_1 A^*) B^*) \\ &\subseteq B^* \end{aligned} \quad (A.2)$$

$\text{range } g'_1 \subseteq B^*$ implies (by Theorem A.12) that $\text{domain } g'_1 = \text{range } g'_1$, and (by Theorem A.11) that $g'_2 \subseteq g''_2$. \square

A.2.2 Pairing

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

Lemma A.15 ($\langle \cdot, \cdot \rangle_{\text{map}}$ measurability). *If $g_1 : X \rightarrow Y_1$ is \mathcal{A} - \mathcal{B}_1 -measurable and $g_2 : X \rightarrow Y_2$ is \mathcal{A} - \mathcal{B}_2 -measurable, then $\langle g_1, g_2 \rangle_{\text{map}}$ is \mathcal{A} - $(\mathcal{B}_1 \otimes \mathcal{B}_2)$ -measurable.*

Theorem A.16 ($\&\&_{\text{map}}$ measurability). *If $g_1 : X \xrightarrow{\text{map}} Y_1$ is \mathcal{A} - \mathcal{B}_1 -measurable and $g_2 : X \xrightarrow{\text{map}} Y_2$ is \mathcal{A} - \mathcal{B}_2 -measurable, then $g_1 \&\&_{\text{map}} g_2$ is \mathcal{A} - $(\mathcal{B}_1 \otimes \mathcal{B}_2)$ -measurable.*

Proof. Let A_1^* and A_2^* be respectively g_1 's and g_2 's maximal domains. The maximal domain of $g_1 \&\&_{\text{map}} g_2$ is $A^{**} := A_1^* \cap A_2^*$, which is in \mathcal{A} . By definition, $(g_1 \&\&_{\text{map}} g_2) A^{**} = \langle g_1 A^{**}, g_2 A^{**} \rangle_{\text{map}}$, which by Lemma A.15 is \mathcal{A} - $(\mathcal{B}_1 \otimes \mathcal{B}_2)$ -measurable. \square

A.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 A.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 A.18 (ifte_{map} measurability). *If $g_1 : X \xrightarrow{\text{map}} \text{Bool}$, and $g_2 : X \xrightarrow{\text{map}} Y$ and $g_3 : X \xrightarrow{\text{map}} Y$ are respectively \mathcal{A} - $(\mathcal{P} \text{ Bool})$ -measurable and \mathcal{A} - \mathcal{B} -measurable, then $\text{ifte}_{\text{map}} g_1 g_2 g_3$ is \mathcal{A} - \mathcal{B} -measurable.*

Proof. Let A_1^* , A_2^* and A_3^* be g_1 's, g_2 's and g_3 's maximal domains. The maximal domain of $\text{ifte}_{\text{map}} g_1 g_2 g_3$ is A^{**} , defined by

$$\begin{aligned} A_2^{**} &:= A_2^* \cap \text{preimage}(g_1 A_1^*) \{\text{true}\} \\ A_3^{**} &:= A_3^* \cap \text{preimage}(g_1 A_1^*) \{\text{false}\} \\ A^{**} &:= A_2^{**} \uplus A_3^{**} \end{aligned} \tag{A.3}$$

Because $\text{preimage } (g_1 \ A_1^*) \ B \in \mathcal{A}$ for any $B \subseteq \text{Bool}$, $A^{**} \in \mathcal{A}$. By definition,

$$\begin{aligned} \text{ifte}_{\text{map}} \ g_1 \ g_2 \ g_3 \ A^{**} = & \text{ let } g'_1 := g_1 \ A^{**} \\ & g'_2 := g_2 \ (\text{preimage } g'_1 \ \{\text{true}\}) \\ & g'_3 := g_3 \ (\text{preimage } g'_1 \ \{\text{false}\}) \\ & \text{in } g'_2 \uplus_{\text{map}} g'_3 \end{aligned} \tag{A.4}$$

By hypothesis, g'_1 , g'_2 and g'_3 are measurable mappings. By Theorem 7.44 (mapping arrow restriction), g'_2 and g'_3 have disjoint domains. Apply Lemma A.17. \square

A.2.4 Laziness

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

Theorem A.19 (measurability of \emptyset). *For any σ -algebras \mathcal{A} and \mathcal{B} , the empty mapping \emptyset is \mathcal{A} - \mathcal{B} -measurable.*

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

Theorem A.20 (measurability under lazy_{map}). *Let $g : 1 \Rightarrow (X \rightsquigarrow Y)$. If $g \ 0$ is \mathcal{A} - \mathcal{B} -measurable, then $\text{lazy}_{\text{map}} \ g$ is \mathcal{A} - \mathcal{B} -measurable.*

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

$$\text{lazy}_{\text{map}} \ g \ A^{**} = \text{if } (A^{**} = \emptyset) \ \emptyset \ (g \ 0 \ A^{**}) \tag{A.5}$$

If $A^{**} = \emptyset$, then $\text{lazy}_{\text{map}} \ g \ A^{**} = \emptyset$; apply Theorem A.19. If $A^{**} \neq \emptyset$, then $\text{lazy}_{\text{map}} \ g = g \ 0$, which is \mathcal{A} - \mathcal{B} -measurable. \square

A.3 Measurable Probabilistic Computations

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

Definition A.21 (measurable mapping* arrow computation). *Let \mathcal{A} and \mathcal{B} be σ -algebras on $(\Omega \times T) \times X$ and Y . A computation $g : X \rightsquigarrow_{\text{map}^*} Y$ is **\mathcal{A} - \mathcal{B} -measurable** if $g j_0$ is an \mathcal{A} - \mathcal{B} -measurable mapping arrow computation.*

Theorem A.22. *If $g : X \rightsquigarrow_{\text{map}^*} Y$ is \mathcal{A} - \mathcal{B} -measurable, then for all $j \in J$, $g j$ is an \mathcal{A} - \mathcal{B} -measurable mapping arrow computation.*

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

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

Definition A.23 (standard σ -algebra). *For a set X used as a type, ΣX denotes its **standard σ -algebra**, which must be defined under the following constraints:*

$$\Sigma \langle X_1, X_2 \rangle = \Sigma X_1 \otimes \Sigma X_2 \quad (\text{A.6})$$

$$\Sigma (J \rightarrow X) = (\Sigma X)^{\otimes J} \quad (\text{A.7})$$

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

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

$$\Sigma \text{ Bool } ::= \mathcal{P} \text{ Bool} \quad (\text{A.8})$$

$$\Sigma T ::= \mathcal{P} T \quad (\text{A.9})$$

Lemma A.24 (measurable mapping arrow lifts). *arr_{map} id , arr_{map} fst and arr_{map} snd are measurable. $\text{arr}_{\text{map}} (\text{const } b)$ is measurable if $\{b\}$ is a measurable set. For all $j \in J$, $\text{arr}_{\text{map}} (\pi j)$ is measurable.*

Corollary A.25 (measurable mapping* arrow lifts). *$\text{arr}_{\text{map}^*}$ id , $\text{arr}_{\text{map}^*}$ fst and $\text{arr}_{\text{map}^*}$ snd are measurable. $\text{arr}_{\text{map}^*} (\text{const } b)$ is measurable if $\{b\}$ is a measurable set. $\text{random}_{\text{map}^*}$ and $\text{branch}_{\text{map}^*}$ are measurable.*

Theorem A.26 (AStore combinators preserve measurability). *Every AStore arrow combinator produces measurable mapping* computations from measurable mapping* computations.*

Proof. AStore's combinators are defined in terms of the base arrow's combinators and arr_{map} fst and arr_{map} snd . \square

Theorem A.27 ($\text{ifte}_{\text{map}^*}^\Downarrow$ measurability). *$\text{ifte}_{\text{map}^*}^\Downarrow$ is measurable.*

Proof. $\text{branch}_{\text{map}^*}$ is measurable, and arr_{map} agrees is measurable by (A.8). \square

We can now prove all nonrecursive programs measurable by induction.

Definition A.28 (finite expression). *A **finite expression** is any expression for which no subexpression is a first-order application.*

Theorem A.29 (all finite expressions are measurable). *For all finite expressions e , $\llbracket e \rrbracket_{\text{map}^*}$ is measurable.*

Proof. By structural induction and the above theorems. \square

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

Theorem A.30 (approximation with finite expressions). *Let $g := \llbracket e \rrbracket_{\text{map}^*}^\Downarrow : X \rightsquigarrow Y$ and $t \in T$. Define $A := (\Omega \times \{t\}) \times X$. There is a finite expression e' for which $\llbracket e' \rrbracket_{\text{map}^*} j_0 A = g j_0 A$.*

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

Theorem A.31 (all probabilistic expressions are measurable). *For all expressions e , $\llbracket e \rrbracket_{\text{map}^*}^\Downarrow$ is measurable.*

Proof. Let $g := \llbracket e \rrbracket_{\text{map}^*}^\downarrow$ and $g' := g \circ_0 ((\Omega \times T) \times X)$. By Corollary 7.51 (correct computation everywhere), $g' = g \circ_0 A^*$ where A^* is g 's maximal domain; thus we need only show g' is a measurable mapping.

By Theorem 7.44 (mapping arrow restriction),

$$g' = \bigcup_{t \in T} g \circ_0 ((\Omega \times \{t\}) \times X) \quad (\text{A.10})$$

By Theorem A.30 (approximation with finite expressions), for every $t \in T$, there is a finite expression whose interpretation agrees with g on $(\Omega \times \{t\}) \times X$. Therefore, by Theorem A.29 (all finite expressions are measurable), $g \circ_0 ((\Omega \times \{t\}) \times X)$ is a measurable mapping. By Theorem 7.44 (mapping arrow restriction), they have disjoint domains. By Lemma A.17 (union of measurable mappings), their union is measurable. \square

Theorem A.31 remains true when $\llbracket \cdot \rrbracket_a$ is extended with any rule whose right side is measurable, including rules for real arithmetic, equality, inequality and limits. More generally, any continuous or (countably) piecewise continuous function can be made available as a language primitive, as long as its domain's and codomain's standard σ -algebras are generated from their topologies.

It is not difficult to compose $\llbracket \cdot \rrbracket_a$ with another semantic function that defunctionalizes lambda expressions. Thus, the interpretations of all expressions in higher-order languages are measurable.

A.4 Measurable Projections

If $g := \llbracket e \rrbracket_{\text{map}^*}^\downarrow : X \rightsquigarrow_{\text{map}^*} Y$, the probability of a measurable output set $B \in \Sigma Y$ is

$$P(\text{image}(\text{fst} \ggg \text{fst}) \cap \text{preimage}(g \circ_0 A^*) B) \quad (\text{A.11})$$

Unfortunately, projections are generally not measurable. Fortunately, for interpretations of programs $\llbracket p \rrbracket_{\text{map}^*}^\downarrow$, for which $X = \{\langle \rangle\}$, we have a special case.

Theorem A.32 (measurable finite projections). *Let $A \in \Sigma \langle X_1, X_2 \rangle$. If X_2 is at most countable and $\Sigma X_2 = \mathcal{P} X_2$, then $\text{image fst } A \in \mathcal{A}_1$.*

Proof. Because $\Sigma X_2 = \mathcal{P} X_2$, A is a countable union of rectangles of the form $A_1 \times \{a_2\}$, where $A_1 \in \Sigma X_1$ and $a_2 \in X_2$. Because image fst distributes over unions, $\text{image fst } A$ is a countable union of sets in ΣX_1 . \square

Theorem A.33. *Let $g : X \xrightarrow{\sim}_{\text{map}^*} Y$ be measurable. If X is at most countable and $\Sigma X = \mathcal{P} X$, for all $B \in \Sigma Y$, $\text{image}(\text{fst} \ggg \text{fst})(\text{preimage}(g \circ A^*) B) \in \Sigma \Omega$.*

Proof. T is countable and $\Sigma T = \mathcal{P} T$ by (A.9); apply Theorem A.32 twice. \square

In particular, for $\llbracket p \rrbracket_{\text{map}^*}^\downarrow : \{\langle \rangle\} \xrightarrow{\sim}_{\text{map}^*} Y$, the probabilities of ΣY are well-defined.

Appendix B

Sampling Theorems

This chapter contains proofs of measure-theoretic theorems stated in Chapter 8.

B.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.

B.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 B.1 (measure). *A partial function $m : \text{Set } X \rightarrow [0, +\infty]$ with domain $\mathcal{A} := \text{domain } m$ is a **measure** if*

- \mathcal{A} is a σ -algebra
- $m \emptyset = 0$
- It is σ -**additive**: for any disjoint collection $A : \mathbb{N} \Rightarrow \text{Set } X$ of sets in \mathcal{A} ,

$$m \left(\bigcup_{n \in \mathbb{N}} A_n \right) = \sum_{n \in \mathbb{N}} m(A_n) \quad (\text{B.1})$$

From here on, we rely again on the notion of a set X 's standard σ -algebra Σ_X , and assume the domain of a measure $m : \text{Set } X \rightarrow [0, 1]$ is Σ_X .

We will need to distinguish three kinds of measures.

Definition B.2 (probability, finite, and σ -finite measures). *A measure $m : \text{Set } X \rightarrow [0, +\infty]$ may be*

- A **probability measure** if $m(X) = 1$.
- A **finite measure** if $m(X) < +\infty$.
- A **σ -finite measure** if there is a collection $A : \mathbb{N} \Rightarrow \Sigma_X$ such that $m(A_n) < +\infty$ for all $n \in \mathbb{N}$, and $\bigcup_{n \in \mathbb{N}} A_n = X$.

Trivially, probability measures are also finite measures, which in turn are also σ -finite.

A ubiquitous example of a σ -finite measure is **Lebesgue measure**,¹ which maps sets of \mathbb{R}^n (for $n \geq 1$) 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. $\mathbb{R} = \bigcup_{n \in \mathbb{N}} [-n, n]$.

Counting measure simply returns the cardinality of a set. If X is countable and $m : \text{Set } X \rightarrow [0, +\infty]$ is counting measure, then m is σ -finite. If X is finite, m is finite.

Image measure defines measures over the outputs of functions in terms of measures over their inputs.

Definition B.3 (image measure). *Let $m : \text{Set } X \rightarrow [0, +\infty]$ be a measure and $g : X \rightarrow Y$ be measurable. Then g 's **image measure** with respect to m is $m' : \text{Set } Y \rightarrow [0, +\infty]$, defined by $m'(B) = m(\text{preimage } g(B))$.*

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.

¹Pronounced “lehBEG,” and named after French mathematician Henri Lebesgue.

Definition B.4 (almost everywhere). A measurable predicate $p?$ holds **almost everywhere** with respect to measure $m : \text{Set } X \rightarrow [0, +\infty]$ when it holds on the complement of a null set:

$$\begin{aligned} \text{ae?} : (\text{Set } X \rightarrow [0, +\infty]) &\Rightarrow (X \rightarrow \text{Bool}) \Rightarrow \text{Bool} \\ \text{ae? } m \ p? &:= m(\text{preimage } p? \ \{\text{false}\}) = 0 \end{aligned} \tag{B.2}$$

If m is a probability measure, $\text{ae? } m \ p?$ is equivalent to $m(\text{preimage } p? \ \{\text{true}\}) = 1$, or to $p?$ holding on a set of measure 1. If m is a finite measure, it is equivalent to $p?$ holding on a set of measure $m(X)$. If m is any other kind of measure, $\text{ae? } m \ p?$ must be determined using null sets. If we were to say $p?$ holds almost everywhere when $m(\text{preimage } p? \ \{\text{true}\}) = m(X) = +\infty$, we would have to say $\lambda x \in \mathbb{R}. x > 0$ holds almost everywhere with respect to Lebesgue measure.

In this chapter, we are most interested in almost-everywhere equality of mappings.

Definition B.5 (almost-everywhere equality). Two total mappings are **equal almost everywhere** with respect to a measure $m : \text{Set } X \rightarrow [0, +\infty]$ when they are not equal only on a null set, or **ae-equal?** $m \ g_1 \ g_2$ where

$$\begin{aligned} \text{ae-equal?} : (\text{Set } X \rightarrow [0, +\infty]) &\Rightarrow (X \rightarrow Y) \Rightarrow (X \rightarrow Y) \Rightarrow \text{Bool} \\ \text{ae-equal? } m \ g_1 \ g_2 &:= \text{ae? } m \ \lambda a \in \text{domain } g_1. g_1(a) = g_2(a) \end{aligned} \tag{B.3}$$

From here on, we use the more common “ $g_1 = g_2$ (m -a.e.)” instead of $\text{ae-equal? } m \ g_1 \ g_2$.

B.1.2 Integration

While measure-theoretic integration—called **Lebesgue integration**—is λ_{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 integration² (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}^n .

²Pronounced “REEmahn,” and named after the German mathematician Bernhard Riemann.

- Lebesgue integration with respect to Lebesgue measure is strictly more general than Riemann integration on \mathbb{R}^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

$$\begin{aligned} \text{int} : (X \rightarrow \mathbb{R}) &\Rightarrow (\text{Set } X \rightharpoonup [0, +\infty]) \Rightarrow (\text{Set } X \rightharpoonup [-\infty, +\infty]) \\ \text{int } g \ m &:= \lambda A \in \text{domain } m. \int_A g \ dm \end{aligned} \tag{B.4}$$

Now $\text{int } g \ m$ is an *indefinite* integral of g : another partial function, defined on the domain of m , that returns the definite integral on a given set A . For example, if $g x := x^2$ and $m : \text{Set } \mathbb{R} \rightharpoonup [0, +\infty]$ is Lebesgue measure, $\text{int } g \ m$ measures areas under the curve $y = x^2$.

We can compute areas under the curve $y = x^2$ using Riemann integration:

$$\text{int } g \ m [0, 1] = \int_{[0,1)} g \ dm = \int_0^1 x^2 \ dx = \frac{1^3}{3} - \frac{0^3}{3} = \frac{1}{3} \tag{B.5}$$

Of course, $\text{int } g \ m$ accepts any $A \in \text{domain } m$. Because $\text{domain } m$ is a σ -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 B.6 (real function a.e. equality). *If $m : \text{Set } X \rightharpoonup [0, +\infty]$ is a σ -finite measure and $g_1, g_2 : X \rightarrow \mathbb{R}$ are measurable, then $g_1 = g_2$ (m -a.e) if and only if $\text{int } g_1 \ m = \text{int } g_2 \ m$.*

The type of int might suggest its intended use; in particular, $\text{Set } X \rightharpoonup [-\infty, +\infty]$ is similar to $\text{Set } X \rightharpoonup [0, +\infty]$, which we use as the type of measures.³ We have functionalized

³In fact, $\text{Set } X \rightharpoonup [-\infty, +\infty]$ is the type we would use for *signed* measures if we needed them.

indefinite integration to emphasize that, in this chapter and much of measure-theoretic 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 B.7 (indefinite integration yields measures). *If $g : X \rightarrow [0, +\infty)$ is measurable and $m : \text{Set } X \rightharpoonup [0, +\infty]$ is a measure, then $\text{int } g m$ is a measure.*

For example, if $g : \mathbb{R} \rightarrow [0, +\infty)$ is a probability density function and m is Lebesgue measure on \mathbb{R} , then $\text{int } g m$ is a probability measure.

Lemma B.7 implies there is a function

$$\text{int}^+ : (X \rightarrow [0, +\infty)) \Rightarrow (\text{Set } X \rightharpoonup [0, +\infty]) \Rightarrow (\text{Set } X \rightharpoonup [0, +\infty]) \quad (\text{B.6})$$

which agrees with int for all arguments $g : X \rightarrow [0, +\infty)$. We thus begin defining an algebra of measures and operations on them with 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 B.8. *Let $g_1, g_2 : X \rightarrow [0, +\infty)$ be measurable and $m : \text{Set } X \rightharpoonup [0, +\infty]$ be a measure. Then $\text{int } (g_1 + g_2) m = \text{int } g_1 m + \text{int } g_2 m$.*

Lemma B.9. *Let $g : X \rightarrow [0, +\infty)$ be measurable, $\alpha \geq 0$ and $m : \text{Set } X \rightharpoonup [0, +\infty]$ be a measure. Then $\text{int } (\alpha \cdot g) m = \alpha \cdot \text{int } g m$.*

Lastly, compositions within integrals can be moved into the base measure.

Lemma B.10 (image measure integration). *Let $m : \text{Set } X \rightharpoonup [0, +\infty]$ be a measure, $g_1 : X \rightarrow Y$ and $g_2 : Y \rightarrow \mathbb{R}$ be measurable, and m' be g_1 's image measure with respect to m . For all $B \in \Sigma Y$, $\text{int } (g_2 \circ_{\text{map}} g_1) m (\text{preimage } g_1 B) = \text{int } g_2 m' B$. [XXX: need more conditions]*

B.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 B.11 (absolute continuity). *Given measures $m_1, m_2 : \text{Set } X \rightarrow [0, +\infty]$, we say m_1 is **absolutely continuous** with respect to m_2 if $m_1 \ll m_2$, where*

$$\begin{aligned} (\ll) : (\text{Set } X \rightarrow [0, +\infty]) &\Rightarrow (\text{Set } X \rightarrow [0, +\infty]) \Rightarrow \text{Bool} \\ m_1 \ll m_2 &:= \forall A \in \text{domain } m_2. m_2(A) = 0 \implies m_1(A) = 0 \end{aligned} \tag{B.7}$$

By Definition B.11, $m_1 \ll m_2$ means m_1 has at least as many measure-zero sets as m_2 , and is therefore, in a sense, smaller. If P and Q are probability measures, $P \ll Q$ essentially means P 's support is no larger than Q 's support.

As for integration, for differentiation, we functionalize special notation:

$$\begin{aligned} \text{diff}^+ : (\text{Set } X \rightarrow [0, +\infty]) &\Rightarrow (\text{Set } X \rightarrow [0, +\infty]) \Rightarrow (X \rightarrow [0, +\infty)) \\ \text{diff}^+ m_1 m_2 &:= \frac{dm_1}{dm_2} \end{aligned} \tag{B.8}$$

This returns a **Radon-Nikodým derivative**. Such derivatives are named after the following theorem, which gives circumstances under which $\text{diff}^+ m_1 m_2$ exists, and states that int^+ is the left inverse of diff^+ (with second arguments held constant).

Lemma B.12 (Radon-Nikodým). *If $m_1, m_2 : \text{Set } X \rightarrow [0, +\infty]$ are σ -finite measures and $m_1 \ll m_2$, then $\text{diff}^+ m_1 m_2$ exists and $m_1 = \text{int}^+(\text{diff}^+ m_1 m_2) m_2$.*

The function $\text{diff}^+ m_1 m_2 : X \rightarrow [0, +\infty)$ is often called the *density* of m_1 with respect to m_2 , but we call them *derivatives*, reserving *density* for derivatives with respect to Lebesgue measure. By Lemma B.6, any $g : X \rightarrow [0, +\infty)$ for which $g = \text{diff}^+ m_1 m_2$ (m_2 -a.e.) meets the Radon-Nikodým theorem's conclusion $m_1 = \text{int}^+ g m_2$. We therefore say that Radon-Nikodým derivatives are unique up to equality m_2 -a.e.

By analogy to differential calculus, we should expect diff^+ to be the left inverse of int^+ (with second arguments held constant). It is, up to equality m_2 -a.e.

Lemma B.13. *If $g_1 : X \rightarrow [0, +\infty)$ is measurable and $m_2 : \text{Set } X \rightharpoonup [0, +\infty]$ is a σ -finite measure, then $\text{int}^+ g_1 m_2 \ll m_2$ and $g_1 = \text{diff}^+ (\text{int}^+ g_1 m_2) m_2$ (m_2 -a.e.).*

The preceding two theorems are analogous to the fundamental theorem of calculus.

We should expect differentiation to be positive linear, and it is.

Lemma B.14. *Let $m_1, m_2, m : \text{Set } X \rightharpoonup [0, +\infty]$ be σ -finite measures with $m_1 \ll m$ and $m_2 \ll m$. Then $m_1 + m_2 \ll m$ and $\text{diff}^+ (m_1 + m_2) m = \text{diff}^+ m_1 m + \text{diff}^+ m_2 m$ (m -a.e.).*

Lemma B.15. *Let $m_1, m_2 : \text{Set } X \rightharpoonup [0, +\infty]$ be σ -finite measures with $m_1 \ll m_2$. For all $\alpha \geq 0$ and $\beta > 0$, $\alpha \cdot m_1 \ll \beta \cdot m_2$ and $\text{diff}^+ (\alpha \cdot m_1) (\beta \cdot m_2) = \frac{\alpha}{\beta} \cdot \text{diff}^+ m_1 m_2$ (m -a.e.).*

As in differential calculus, there is a chain rule.

Lemma B.16 (chain rule). *Let $m_1, m_2, m_3 : \text{Set } X \rightharpoonup [0, +\infty]$ be σ -finite measures with $m_1 \ll m_2$ and $m_2 \ll m_3$. Then $m_1 \ll m_3$ and $\text{diff}^+ m_1 m_2 \cdot \text{diff}^+ m_2 m_3 = \text{diff}^+ m_1 m_3$ (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 B.17 (reciprocal rule). *Let $m_1, m_2 : \text{Set } X \rightharpoonup [0, +\infty]$ be σ -finite measures with $m_2 \ll m_1$ and $m_1 \ll m_2$. Then $\text{diff}^+ m_1 m_2 = 1 / \text{diff}^+ m_2 m_1$ (m_1 -a.e. and 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 B.18 (change of measure). *Let $m_1, m_2 : \text{Set } X \rightharpoonup [0, +\infty]$ be σ -finite measures with $m_1 \ll m_2$, and $g : X \rightarrow \mathbb{R}$ be measurable. Then $\text{int } g m_1 = \text{int } (g \cdot \text{diff}^+ m_1 m_2) m_2$.*

Suppose we have a joint and candidate probability densities $p, q : \mathbb{R}^n \rightarrow [0, +\infty)$, and we sample according to q and weight the samples by p / q . The weighted samples represent p if expected values estimated using them are correct; i.e. for all measurable $g : \mathbb{R}^n \rightarrow \mathbb{R}$,

$$\text{int } g P = \text{int } (g \cdot p / q) Q \quad (\text{B.9})$$

where $P := \text{int}^+ p m$ and $Q := \text{int}^+ q m$, and m is Lebesgue measure on \mathbb{R}^n .

The density route to a proof is simple, and requires that q be nonzero everywhere. By definition and Lemma B.13 (diff^+ is a left inverse of int^+), $q = \text{diff}^+ Q m$ (m -a.e.), so by Lemma B.18 (change of measure) with $m_1 = Q$,

$$\begin{aligned} \text{int } (g \cdot p / q) Q &= \text{int } (g \cdot p / q \cdot q) m \\ &= \text{int } (g \cdot p) m \end{aligned} \quad (\text{B.10})$$

which again by Lemmas B.13 and B.18 is $\text{int } g P$.

Taking the measure route demonstrates how to prove more general importance sampling theorems. We again require q to be nonzero everywhere; then

$$\begin{aligned} p / q &= p \cdot 1 / q = \text{diff}^+ P m \cdot 1 / \text{diff}^+ Q m \quad (\text{m-a.e.}) \quad \text{Lemma B.13} \\ &= \text{diff}^+ P m \cdot \text{diff}^+ m Q \quad (\text{m, Q-a.e.}) \quad \text{Lemma B.17} \\ &= \text{diff}^+ P Q \quad (\text{Q-a.e.}) \quad \text{Lemma B.16} \end{aligned} \quad (\text{B.11})$$

Because $g \cdot p / q = g \cdot \text{diff}^+ P Q$ (Q-a.e.),

$$\begin{aligned} \text{int } (g \cdot p / q) Q &= \text{int } (g \cdot \text{diff}^+ P Q) Q \quad \text{Lemma B.6} \\ &= \text{int } g P \quad \text{Lemma B.18} \end{aligned} \quad (\text{B.12})$$

The more general method is this: instead of densities p and q , define measures P and Q , derive $\text{diff}^+ P Q$, and apply Lemma B.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.

B.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 B.19 (transition kernel). *A function $k : X \rightarrow \text{Set } Y \rightharpoonup [0, +\infty]$ is a **transition kernel** when both of the following hold.*

- *For all $a \in X$, $k|_a$ is a measure.*
- *For all $B \in \Sigma Y$, $\lambda a \in X. k|_a B$ is measurable.*

For any measure property, we say k has that property when it holds for all $k|_a$. Therefore, k is a **probability kernel**, **finite kernel**, or **σ -finite kernel** when for all $a \in X$, $k|_a$ is respectively a probability measure, finite measure, or σ -finite measure.

Product models of dependent processes can be built by starting with a probability measure and iteratively extending it using probability kernels.

Lemma B.20 (finite kernel products). *Let $m : \text{Set } X \rightarrow [0, +\infty]$ be a finite measure and $k : X \rightarrow \text{Set } Y \rightharpoonup [0, +\infty]$ be a finite kernel. There exists a unique σ -finite measure $m \times k : \text{Set } \langle X, Y \rangle \rightharpoonup [0, +\infty]$ that is determined by its output on rectangles; i.e. defined by extending the following to a product measure: for all $A \in \Sigma X$ and $B \in \Sigma Y$,*

$$(m \times k)(A \times B) = \text{int}^+ (\lambda a \in X. k|_a B) m A \quad (\text{B.13})$$

If m is a probability measure and k a probability kernel, $m \times k$ is a probability measure.

For example, if $N : \mathbb{R} \rightarrow \text{Set } \mathbb{R} \rightharpoonup [0, 1]$ takes a mean μ and returns a normal probability measure centered on μ with standard deviation 1, then the interpretation of

$$\begin{aligned} X &\sim \text{Normal}(0, 1) \\ Y &\sim \text{Normal}(X, 1) \end{aligned} \quad (\text{B.14})$$

as a measure-theoretic joint distribution is $(N|_0) \times N$.

For the proofs in the next section, we need a way to turn integrals with respect to $m \times k$ measures into nested integrals.

Lemma B.21 (Fubini's for transition kernels). *Let $m : \text{Set } X \rightarrow [0, +\infty]$ be a finite measure and $k : X \rightarrow \text{Set } Y \rightarrow [0, +\infty]$ be a finite kernel. If $g : X \times Y \rightarrow \overline{\mathbb{R}}$ is measurable, and nonnegative or $(m \times k)$ -integrable, then*

$$\begin{aligned} & \text{int } g (m \times k) (X \times Y) \\ &= \text{int } (\lambda a \in X. \text{int } (\lambda b \in Y. g \langle a, b \rangle) (k a) Y) m X \end{aligned} \tag{B.15}$$

B.2 Sampling Proofs

Recall that $\text{subcond } P A' := \lambda A \in \text{domain } P. P(A' \cap A)$.

Theorem B.22 (partitioned importance sampling correctness). *Let X, P, N, s, p , and Q as in Definition 8.25 (partitioned importance sampling) such that $\text{subcond } P(s n) \ll Q n$ for all $n \in N$. Define $P_N : \text{Set } N \rightarrow [0, 1]$ by integrating p with respect to counting measure.*

If $g : X \rightarrow \mathbb{R}$ is a P -integrable mapping, and

$$\begin{aligned} g' : N \times X &\rightarrow \mathbb{R} \\ g' \langle n, a \rangle &:= g a \cdot \frac{1}{p n} \cdot \text{diff}^+ (\text{subcond } P(s n)) (Q n) a \end{aligned} \tag{B.16}$$

then $\text{int } g' (P_N \times Q) (N \times X) = \text{int } g P X$.

Proof. Let $w_1 n := \frac{1}{p n}$ and $w_2 n := \text{diff}^+ (\text{subcond } P(s n)) (Q n)$. Starting from the left side,

$$\begin{aligned} & \text{int } g' (P_N \times Q) (N \times X) \\ &= \text{int } (\lambda \langle n, a \rangle \in N \times X. g a \cdot w_1 n \cdot w_2 n) (P_N \times Q) (N \times X) && \text{Def of } g' \\ &= \text{int } (\lambda n \in N. \text{int } (\lambda a \in X. g a \cdot w_1 n \cdot w_2 n) (Q n) X) P_N N && \text{Lemma B.21} \\ &= \text{int } (\lambda n \in N. \text{int } (g \cdot w_1 n \cdot w_2 n) (Q n) X) P_N N && \text{Lift } (\cdot) \\ &= \text{int } (\lambda n \in N. w_1 n \cdot \text{int } (g \cdot w_2 n) (Q n) X) P_N N && \text{Lemma B.9} \\ &= \text{int } (\lambda n \in N. w_1 n \cdot \text{int } g (\text{subcond } P(s n)) X) P_N N && \text{Def } w_2, \text{ Lemma B.18} \end{aligned} \tag{B.17}$$

Because P_N is defined with respect to counting measure, turn integration into summation:

$$\begin{aligned}
&= \sum_{n \in N} p_n \cdot \frac{1}{p_n} \cdot \text{int } g(\text{subcond } P(s_n)) X && \text{Def of } w_1 \\
&= \sum_{n \in N} \text{int } g(\text{subcond } P(s_n)) X && p_n > 0 \\
&= \sum_{n \in N} \text{int } g P(s_n) && \text{XXX: justify} \\
&= \text{int } g P(\bigcup_{n \in N} (s_n)) && \sigma\text{-additivity} \\
&= \text{int } g P X && \text{Def of } s \quad \square
\end{aligned}$$

When m_1 and m_2 are measures on infinite spaces, it is not clear that $\text{diff}^+ m_1 m_2$ exists or how to compute it. It seems it should exist when m_1 and 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 $\text{diff}^+ (m_1 \times k) (m_2 \times k)$ that says we may ignore k .

Theorem B.23. *Let $m_1, m_2 : \text{Set } X \rightarrow [0, +\infty]$ be finite measures such that $m_1 \ll m_2$, and $k : X \rightarrow \text{Set } Y \rightarrow [0, +\infty]$ be a finite kernel. Then $m_1 \times k \ll m_2 \times k$ and $\text{diff}^+ (m_1 \times k) (m_2 \times k) = (\lambda \langle a, b \rangle \in X \times Y. \text{diff}^+ m_1 m_2 a) (m_2 \times k\text{-a.e.})$.*

Proof. XXX: prove absolute continuity

Let $A \in \Sigma X$ and $B \in \Sigma Y$. Integrating the left-hand side,

$$\begin{aligned}
&\text{int } (\text{diff}^+ (m_1 \times k) (m_2 \times k)) (m_2 \times k) (A \times B) && (B.18) \\
&= (m_1 \times k) (A \times B) && \text{Lemma B.12}
\end{aligned}$$

Integrating the right-hand side,

$$\begin{aligned}
& \text{int} (\lambda \langle a, b \rangle \in X \times Y. \text{diff}^+ m_1 m_2 a) (m_2 \times k) (A \times B) && \text{(B.19)} \\
&= \text{int} (\lambda a \in X. \text{int} (\lambda b \in Y. \text{diff}^+ m_1 m_2 a) (k a) B) m_2 A && \text{Lemma B.21} \\
&= \text{int} (\text{diff}^+ m_1 m_2 \cdot \lambda a \in X. \text{int} (\lambda b \in Y. 1) (k a) B) m_2 A && \text{Lemma B.9, Lift } (\cdot) \\
&= \text{int} (\lambda a \in X. k a B) m_1 A && \text{Lemma B.18} \\
&= (m_1 \times k) (A \times B) && \text{Lemma B.20}
\end{aligned}$$

Therefore, because $m_1 \times k$ is uniquely defined by its output on all such $A \times B$,

$$\begin{aligned}
& \text{int} (\text{diff}^+ (m_1 \times k) (m_2 \times k)) (m_2 \times k) && \text{(B.20)} \\
&= \text{int} (\lambda \langle a, b \rangle \in X \times Y. \text{diff}^+ m_1 m_2 a) (m_2 \times k)
\end{aligned}$$

Apply Lemma B.6 (real function a.e. equality). \square

It is not hard to extend the preceding 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 $\Omega := J \rightarrow [0, 1]$ from tree indexes to reals.

One solution is to define an injective transformation g from any $\omega \in \Omega$ to a pair $\langle \omega_{\text{fin}}, \omega_{\text{inf}} \rangle$, where ω_{fin} is a finite substructure of ω and ω_{inf} is the rest of it, and apply Theorem B.23. The proof is easier to do first in generality, without specifying the structure of ω , requiring the substructure to be finite, or requiring the pairs to contain projections.

Theorem B.24. *Let $\mu_1, \mu_2 : \text{Set } Z \rightharpoonup [0, +\infty]$ be σ -finite measures. If there exist finite measures $m_1, m_2 : \text{Set } X \rightharpoonup [0, +\infty]$ such that $m_1 \ll m_2$, a finite kernel $k : X \rightarrow \text{Set } Y \rightharpoonup [0, +\infty]$, and an injective, measurable function $g : Z \rightarrow X \times Y$ such that for all $D \in \Sigma \langle X, Y \rangle$,*

$$\begin{aligned}
(m_1 \times k) D &= \mu_1 (\text{preimage } g D) \\
(m_2 \times k) D &= \mu_2 (\text{preimage } g D)
\end{aligned} \tag{B.21}$$

then $\mu_1 \ll \mu_2$ and $\text{diff}^+ \mu_1 \mu_2 = \lambda z \in Z. \text{diff}^+ m_1 m_2 (\text{fst } (g z))$ (μ_2 -a.e.).

Proof. Suppose m_1, m_2, k and g as stated. XXX: prove absolute continuity

Let $C \in \Sigma Z$ and $D := \text{image } g C$; then

$$\begin{aligned}
& \text{int}(\text{diff}^+ \mu_1 \mu_2) \mu_2 C && \text{(B.22)} \\
&= \mu_1 C && \text{Lemma B.12} \\
&= (m_1 \times k) D && \text{Def } (m_1 \times k), \text{ injectivity} \\
&= \text{int}(\text{diff}^+ (m_1 \times k) (m_2 \times k)) (m_2 \times k) D && \text{Lemma B.12} \\
&= \text{int}(\lambda \langle a, b \rangle \in X \times Y. \text{diff}^+ m_1 m_2 a) (m_2 \times k) D && \text{Theorem B.23} \\
&= \text{int}((\lambda \langle a, b \rangle \in X \times Y. \text{diff}^+ m_1 m_2 a) \circ_{\text{map}} g) \mu_2 C && \text{Lemma B.10} \\
&= \text{int}(\lambda z \in Z. \text{diff}^+ m_1 m_2 (\text{fst } (g z))) \mu_2 C && \text{Def of } \circ_{\text{map}}
\end{aligned}$$

Apply Lemma B.6 (real function a.e. equality). \square

Thus, two measures μ_1 and μ_2 on infinite structures that can be decomposed into products $m_1 \times k$ and $m_2 \times k$ such that $\text{diff}^+ m_1 m_2$ exists—using any injective transformation—have a Radon-Nikodým derivative that can be uniquely defined in terms of $\text{diff}^+ m_1 m_2$.

Application to infinite binary trees mostly requires defining the transformation.

Theorem B.25. Let $J' \subseteq J$ be finite, and define $X := J' \rightarrow [0, 1]$ and $Y := (J \setminus J') \rightarrow [0, 1]$. Let $P', Q' : \text{Set } X \rightarrow [0, 1]$ be finite measures such that $P' \ll Q'$, and let $k : X \rightarrow \text{Set } Y \rightarrow [0, 1]$ be a finite kernel. Let $g : \Omega \rightarrow X \times Y$ be defined by $g \omega = \langle \text{restrict } \omega J', \text{restrict } \omega (J \setminus J') \rangle$. If $P, Q : \text{Set } \Omega \rightarrow [0, 1]$ are defined so that for all $\Omega' \in \Sigma \Omega$,

$$\begin{aligned}
P \Omega' &= (P' \times k) (\text{image } g \Omega') && \text{(B.23)} \\
Q \Omega' &= (Q' \times k) (\text{image } g \Omega')
\end{aligned}$$

then $P \ll Q$ and $\text{diff}^+ P Q = \lambda \omega \in \Omega. \text{diff}^+ P' Q' (\text{restrict } \omega J')$ (Q -a.e.).

Proof. The inverse of g is $g^{-1} : X \times Y \rightarrow \Omega$, defined by

$$g^{-1} \langle \omega_{\text{fin}}, \omega_{\text{inf}} \rangle = \lambda j \in J. \text{if } (j \in J') (\omega_{\text{fin}} j) (\omega_{\text{inf}} j) \quad (\text{B.24})$$

Thus $(P' \times k) D = P$ (preimage $g D$) for all $D \in \Sigma(X, Y)$; similarly for $(Q' \times k) D$. Apply Theorem B.24. \square

In particular, if additionally P' and Q' can be defined by densities $p : (J' \rightarrow [0, 1]) \rightarrow [0, +\infty)$ and $q : (J' \rightarrow [0, 1]) \rightarrow [0, +\infty)$, then

$$\text{diff}^+ P Q = \lambda \omega \in \Omega. \frac{p(\text{restrict } \omega J')}{q(\text{restrict } \omega J')} \quad (\text{Q-a.e.}) \quad (\text{B.25})$$

References

- [1] Haskell 98 language and libraries, the revised report, December 2002. URL <http://www.haskell.org/onlinereport/>.
- [2] Stephen Abbott. *Understanding Analysis*. Springer, 2001.
- [3] Peter Aczel. An introduction to inductive definitions. *Studies in Logic and the Foundations of Mathematics*, 90:739–782, 1977.
- [4] G. Amato and F. Scozzari. The abstract domain of parallelotopes. *Electronic Notes in Theoretical Computer Science*, 287:17–28, November 2012.
- [5] Robert J. Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5:614–630, 1961.
- [6] Bruno Barras. Sets in Coq, Coq in sets. *Journal of Formalized Reasoning*, 3(1), 2010.
- [7] C. Berline and K. Grue. A κ -denotational semantics for Map Theory in ZFC+SI. *Theoretical Computer Science*, 179(1–2):137–202, 1997.
- [8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004. URL <http://www.labri.fr/publications/13a/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] Lawrence C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11:353–389, 1993.
- [43] Lawrence C. Paulson. Set theory for verification: II. Induction and recursion. *Journal of Automated Reasoning*, 15:167–215, 1995.
- [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 λ_{ZFC} . In *Impl. and Appl. of Functional Languages*, 2010.
- [49] Neil Toronto and Jay McCarthy. Computing in Cantor’s paradise with λ_{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] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *ACM SIGGRAPH*, pages 65–76, 1997.
- [55] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*. 2001.
- [56] Benjamin Werner. Sets in types, types in sets. In *TACS'97*, pages 530–546, 1997.
- [57] 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.
- [58] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Artificial Intelligence and Statistics*, 2011.