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

January 2014

ABSTRACT

Trustworthy, Useful Languages for Probabilistic Modeling and Inference

Neil Toronto
Department of Computer Science, BYU
Doctor of Philosophy

XXX: rewrite (was lifted from proposal)

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

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

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

Keywords: Probability, Domain-Specific Languages, Semantics

ACKNOWLEDGMENTS

XXX: write

Table of Contents

List of Figures

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

Agatha Christie, *The Mirror Crack'd*

Chapter 1

Thesis Statement

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

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

Such languages should be made to meet a mathematical specification. The reason is simple: if a probabilistic language is made to always meet its maker's expectations, it is almost certainly faulty.

XXX: other reasons

XXX: short segue to...

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

1.1 Statement Terms

Funcional Programming Theory. Blah blah blah.

Blah.

Measure-Theoretic Probability. Blah blah blah.

Blah.

Trustworthy. Blah blah blah.

Blah.

Useful. Blah blah blah.

Blah.

Constructive Probabilitic Modeling. Blah blah blah.

Blah.

Probabilistic Inference. Blah blah blah.

Blah.

1.2 Proof and Supporting Evidence

Semantics. Blah blah blah.

Blah.

Properties. Blah blah blah.

Blah.

Test Cases. Blah blah blah.

Blah.

Chapter 2

Background

The most difficult part of doing work in an emerging cross-disciplinary area is choosing a side and tutoring that side on the background knowledge taken for granted by the other. Finding a viable approach to bridge the two sides is a close second, followed by building the formalisms that bridge them and deciding which theorems to prove. Proving theorems and writing implementations are not only similar tasks, but are just as simple compared to convincing a reader that it is worthwhile to learn new notation and think in a different way.

We cannot choose both sides. In an emerging cross-disciplinary area, there are very few readers who already have enough background in both sides to evaluate the work.

We cannot choose *neither* side, usually. Hardly any respectable publication venue is sufficiently external to both sides but still interested, nor allows enough space to tutor everyone. There is one major exception: the university publication of a dissertation.

Dissertations are intended for a more general audience than either side, and have formats that allow the necessary tutorials. They are evaluated by a readership with at least five members who are, if not *interested*, at least *obligated*. Unfortunately, we must deal with the fact that dissertations written about emerging cross-disciplinary areas are assembled from published and publishable work that is targeted at just one side.

In our work, the two sides are Bayesian practice and functional programming theory. The implicit background knowledge includes:

- In Bayesian practice, probability densities and manipulation rules, basic statistics, Bayesian modeling and philosophy, integration, and inference methods.

- In functional programming, λ-calculus, big-step operational semantics, denotational semantics, categorical semantics, and abstract interpretation.

The greatest commonality in the two sides is the attitude with which they approach modeling. While Bayesian practitioners model processes and functional programming researchers model languages, both approach their tasks methodically, and both create models in which every entity they want to reason about is represented explicitly—sometimes painfully so. Both sides trust their explicit models to lead to more reliable artifacts and repeatable results than models created by other means.

Therefore, for us, choosing a side to target for publication comes down to determining which side's venues will tolerate the precision necessary to explicitly define our models. At this stage, in which we almost exclusively model languages, it is clear that we must choose functional programming. Thus, most of this dissertation assumes readers are functional programming researchers, and tutors them in the necessary Bayesian background.

For readers who are not functional programming researchers, we present this overview of the relevant functional programming theory, and beg their indulgence as we use it to model languages with enough precision to make our artifacts reliable and our results repeatable.

We assume readers know basic computer science theory, including propositional logic, relations, functions, proof by induction, context-free grammars, and nondeterminism.

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**.

$$\begin{aligned} e &::= x \mid e\ e \mid \lambda x.\, e \\ x &::= \text{[variable names]} \end{aligned} \tag{2.1}$$

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 Turing machines, the pure λ-calculus is a minimal universal model of computation. Also like Turing machines, it would be quite painful to program with it. Unlike Turing machines, it is easy to get something practical by extending the pure λ-calculus with values such as pairs and numbers, and a few primitive functions to operate on them.

In most programming languages, the language implementation defines the meaning of function calls. Typically, the implementation pushes arguments onto a stack, copies them to a heap, or copies them to registers. It then stores a return address, jumps to a function address, executes the function body, and jumps back to the return address.

In the pure λ-calculus and most of its extensions, there are no stacks, heaps, registers, addresses or jumps. Function application is defined entirely in terms of substitution, as in algebra. For example, suppose *hypot* is a term in a λ-calculus extended with real numbers, defined by

$$\mathit{hypot} \;=\; \lambda x.\, \lambda y.\, \sqrt{x^2 + y^2} \tag{2.2}$$

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

actual arguments:

$$\begin{aligned} \mathit{hypot}\ 3\ 4 &= \left(\lambda x.\, \lambda y.\, \sqrt{x^2 + y^2}\right)\ 3\ 4 \\ &= \left(\left(\lambda x.\, \left(\lambda y.\, \sqrt{x^2 + y^2}\right)\right)\ 3\right)\ 4 \\ &\equiv \left(\lambda y.\, \sqrt{3^2 + y^2}\right)\ 4 \\ &\equiv \sqrt{3^2 + 4^2} \end{aligned} \tag{2.3}$$

The two equivalences at the end of (2.3) 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 stacks, heaps, registers, addresses and jumps. 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**[1] **indexes**. One form of the pure λ-calculus with De Bruijn indexes is

$$\begin{aligned} e &::= \mathsf{env}\ n \mid e\ e \mid \lambda.\, e \\ n &::= 0 \mid 1 \mid 2 \mid \cdots \end{aligned} \tag{2.4}$$

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 with real numbers:

$$\mathit{hypot} = \lambda.\, \lambda.\, \sqrt{(\mathsf{env}\ 1)^2 + (\mathsf{env}\ 0)^2} \tag{2.5}$$

Here, $\mathsf{env}\ 1$ (which was previously x) refers to the outer lambda's argument and $\mathsf{env}\ 0$ refers

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

to the inner lambda's argument. Reducing an application of *hypot* proceeds this way:

$$\begin{aligned} \mathit{hypot}\ 3\ 4 &= \left(\lambda.\,\lambda.\,\sqrt{(\mathsf{env}\ 1)^2 + (\mathsf{env}\ 0)^2}\right)\ 3\ 4 \\ &\equiv \left(\lambda.\,\sqrt{3^2 + (\mathsf{env}\ 0)^2}\right)\ 4 \\ &\equiv \sqrt{3^2 + 4^2} \end{aligned} \tag{2.6}$$

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

$$\mathit{zero} = \lambda x.\,0 \tag{2.7}$$

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

$$\begin{aligned} \mathit{zero}\ (1\ /\ 0) &= (\lambda x.\,0)\ (1\ /\ 0) \\ &\equiv 0 \end{aligned} \tag{2.8}$$

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 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 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:

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

(a) A grammar to define sets E and V

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

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

Figure 2.1: A big-step operational semantics for a simple addition language.

- **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

$$inc \ = \ \lambda x.\, x + 1 \tag{2.9}$$

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 disseration and it is a big-step semantics, we concentrate on big-step in this overview.

Figure 2.1 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 **inference rules** in the form

$$\frac{premise_1 \quad premise_2 \quad \cdots}{conclusion}\ (\text{name}) \tag{2.10}$$

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 2.1b means "for all $e_1, e_2 \in E$ and $v_1, v_2 \in V$, if e_1 reduces to v_1 and e_2 reduces to v_2, then $\mathsf{add}\ e_1\ e_2$ reduces to $v_1 + v_2$." The (val) rule means "for all $v \in V$, v reduces to v" or equivalently, "for all $v \in V$, *true* implies v reduces to v."

The reduction relation "$\Downarrow$" is defined as the *smallest* subset of $E \times V$ for which the inference 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 inference rules.

Inference 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 inference rules in Figure 2.1b to compute the value of $\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90$. We start by writing it as a conclusion without premises:

$$\frac{}{(\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.11}$$

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

$$\frac{\overline{(\mathsf{add}\ 4\ 5) \Downarrow v_2} \quad \overline{90 \Downarrow v_3}}{(\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.12}$$

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

$$\frac{\dfrac{\overline{4 \Downarrow v_4} \quad \overline{5 \Downarrow v_5}}{(\mathsf{add}\ 4\ 5) \Downarrow v_2} \quad \overline{90 \Downarrow v_3}}{(\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.13}$$

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 2.2: Racket implementation of the semantics defined in Figure 2.1.

them and recursively fill in the conclusions:

$$\dfrac{\dfrac{\overline{4 \Downarrow 4} \quad \overline{5 \Downarrow 5}}{(\mathsf{add}\ 4\ 5) \Downarrow v_2} \quad \overline{90 \Downarrow v_3}}{(\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90) \Downarrow v_1} \Longrightarrow \dfrac{\dfrac{\overline{4 \Downarrow 4} \quad \overline{5 \Downarrow 5}}{(\mathsf{add}\ 4\ 5) \Downarrow 9} \quad \overline{90 \Downarrow 90}}{(\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90) \Downarrow v_1} \Longrightarrow \dfrac{\dfrac{\overline{4 \Downarrow 4} \quad \overline{5 \Downarrow 5}}{(\mathsf{add}\ 4\ 5) \Downarrow 9} \quad \overline{90 \Downarrow 90}}{(\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90) \Downarrow 99} \tag{2.14}$$

Thus, the rightmost derivation tree in (2.14) is a proof that $(\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90) \Downarrow 99$.

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

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

Figure 2.2 gives a Racket implementation of "$\Downarrow$" in Figure 2.1b. The implementation defines a structure type `add` to model add expressions, and uses Racket's built-in big integers to model V. Computation recursively interprets expressions, and proceeds similarly to the derivation tree construction in (2.11) through (2.14). As an example of use, at DrRacket's

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

(a) A grammar to define sets E and V

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

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

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

Figure 2.3: Big-step operational semantics for a language with nondeterministic choice.

Read-Eval-Print Loop (REPL), we get

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

as expected.

Figure 2.3 extends the present example language with nondeterministic choice, which results in a reduction relation that is *not* a function. The culprit is the new rule (choose), with which we can match a single term to multiple conclusions. For example, suppose we want to use the inference rules in Figure 2.3b to compute the value of $\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90$. We start as before, by writing it as a conclusion without premises:

$$\frac{}{(\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.15}$$

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

$$\frac{\overline{(\mathsf{choose}\ 4\ 5) \Downarrow v_2} \quad \overline{90 \Downarrow v_3}}{(\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.16}$$

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

$$\frac{\dfrac{\overline{4 \Downarrow v_4}}{(\mathsf{choose}\ 4\ 5) \Downarrow v_2} \quad \overline{90 \Downarrow v_3}}{(\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90) \Downarrow v_1} \qquad \frac{\dfrac{\overline{5 \Downarrow v_5}}{(\mathsf{choose}\ 4\ 5) \Downarrow v_2} \quad \overline{90 \Downarrow v_3}}{(\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.17}$$

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

$$\begin{aligned} [\![v]\!] &= v \\ [\![\mathsf{add}\ e_1\ e_2]\!] &= [\![e_1]\!] + [\![e_2]\!] \end{aligned}$$

(a) blah

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

(b) blah

Figure 2.4: blah

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

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

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

2.3 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. mathematics).

Figure 2.4a defines a denotational semantics for the addition language without choose

by defining a semantic function $[\![\cdot]\!] : E \to \mathbb{N}$. The double square brackets are simply a different application syntax: they connote nothing mathematically, but serve as a visual cue to read applications of the semantic function as "the meaning of" or "the denotation of." For example, the meaning of add (add 4 5) 90 is

$$\begin{aligned} [\![\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90]\!] &= [\![\mathsf{add}\ 4\ 5]\!] + [\![90]\!] \\ &= ([\![4]\!] + [\![5]\!]) + [\![90]\!] \\ &= (4 + 5) + 90 \\ &= 99 \end{aligned} \tag{2.18}$$

The recursion in a semantic function precisely follows the grammar of a language, and gives meaning to terms *independently of their context*. To describe these properties in one word, we say semantic functions are **compositional**. Constraining semantic functions be compositional allows most proofs of program properties to be done by structural induction, as we will demonstrate shortly.

When the results of applying $[\![\cdot]\!]$ are computable, because it is compositional, it is often easy to implement it as local syntax transformation or compilation. Figure 2.4b shows a Racket implementation of $[\![\cdot]\!]$ as a transformation from meaningless Racket syntax (an `add` function does not exist) to runnable Racket syntax. The syntax transformer is more or less a transcription of the semantic function, 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 terminal.

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))
#'(%app + (%app + '4 '5) '90))
```

$$
\begin{aligned}
[\![\cdot]\!] &: E \to \mathcal{P}\ \mathbb{N} \\
[\![v]\!] &= \{v\} \\
[\![\mathsf{add}\ e_1\ e_2]\!] &= \{v_1 + v_2 \mid v_1 \in [\![e_1]\!], v_2 \in [\![e_2]\!]\} \\
[\![\mathsf{choose}\ e_1\ e_2]\!] &= [\![e_1]\!] \cup [\![e_2]\!]
\end{aligned}
$$

Figure 2.5: blah

The %app in the result is Racket's application operator, which is usually implicit.

Figure 2.5 defines a compositional function $[\![\cdot]\!] : E \to \mathcal{P}\ \mathbb{N}$, which transforms the addition language with nondeterministic choice into sets of natural numbers. For example, the meaning of 4 is $\{4\}$, the meaning of choose 4 5 is $\{4\} \cup \{5\} = \{4, 5\}$, and the meaning of add (choose 4 5) 90 is

$$
\begin{aligned}
[\![\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90]\!] &= \{v_1 + v_2 \mid v_1 \in [\![\mathsf{choose}\ 4\ 5]\!], v_2 \in [\![90]\!]\} \\
&= \{v_1 + v_2 \mid v_1 \in ([\![4]\!] \cup [\![5]\!]), v_2 \in [\![90]\!]\} \\
&= \{v_1 + v_2 \mid v_1 \in (\{4\} \cup \{5\}), v_2 \in \{90\}\} \\
&= \{v_1 + v_2 \mid v_1 \in \{4, 5\}, v_2 \in \{90\}\} \\
&= \{4 + 90, 5 + 90\} \\
&= \{94, 95\}
\end{aligned}
\tag{2.19}
$$

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

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

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

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

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

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

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

Proof of the inductive case $e = \mathsf{choose}\ e_1\ e_2$ is similar to the preceeding (though each "$\Longleftrightarrow$" direction has in inner case for nondeterministic choice) and is left as an exercise. □

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

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

2.4 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**.[2] In semantics, the category is comprised of a collection of objects called **computations** (i.e. possible program meanings) and operations on them called **combinators**. A category may seem like an object-oriented concept, but there is not necessarily any encapsulation, data hiding or inheritance.

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

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

have found very useful are **monads**.[3] The **set monad** operates on set-valued computations and is defined by these two combinators:

$$\begin{aligned} return_{set}\ v &= \{v\} \\ bind_{set}\ A\ f &= \bigcup_{v \in A} f\ v \end{aligned} \tag{2.20}$$

Evidently, we should expect $[\![v]\!]_{set} = return_{set}\ v = \{v\}$. 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 $[\![\mathsf{add}\ e_1\ e_2]\!]$ into an indexed union (as in $bind_{set}$) makes its use clear:

$$\begin{aligned} [\![\mathsf{add}\ e_1\ e_2]\!] &= \{v_1 + v_2 \mid v_1 \in [\![e_1]\!], v_2 \in [\![e_2]\!]\} \\ &= \bigcup_{v_1 \in [\![e_1]\!]} \bigcup_{v_2 \in [\![e_2]\!]} \{v_1 + v_2\} \\ &\equiv \bigcup_{v_1 \in [\![e_1]\!]} \bigcup_{v_2 \in [\![e_2]\!]} return_{set}\ (v_1 + v_2) \\ &\equiv \bigcup_{v_1 \in [\![e_1]\!]} bind_{set}\ [\![e_2]\!]\ (\lambda v_2.\ return_{set}\ (v_1 + v_2)) \\ &\equiv bind_{set}\ [\![e_1]\!]\ (\lambda v_1.\ bind_{set}\ [\![e_2]\!]\ (\lambda v_2.\ return_{set}\ (v_1 + v_2))) \end{aligned} \tag{2.21}$$

Thus, we expect $[\![\mathsf{add}\ e_1\ e_2]\!]_{set} = bind_{set}\ [\![e_1]\!]_{set}\ (\lambda v_1.\ bind_{set}\ [\![e_2]\!]_{set}\ (\lambda v_2.\ return_{set}\ (v_1 + v_2)))$. Finally, we need to extend the set monad with an operation for choose expressions. We define

$$merge_{set}\ A_1\ A_2 = A_1 \cup A_2 \tag{2.22}$$

so that $[\![\mathsf{choose}\ e_1\ e_2]\!]_{set} = merge_{set}\ [\![e_1]\!]_{set}\ [\![e_2]\!]_{set}$.

In Figure 2.6, guided by our expectations for $[\![\cdot]\!]_{set}$, we define a categorical semantics for the addition language with choose, by defining a semantic function $[\![\cdot]\!]_a$ parameterized on a target monad a. The parameterized function M_a returns the monad's computations. If $M_{set}\ X = \mathcal{P}\ X$, then $[\![\cdot]\!]_{set} : E \to M_{set}\ \mathbb{N}$ is equivalent to $[\![\cdot]\!] : E \to \mathcal{P}\ \mathbb{N}$ as defined in Figure 2.5, as expected.

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

$$
\begin{aligned}
[\![\cdot]\!]_a &: E \to M_a\ \mathbb{N} \\
[\![v]\!]_a &= return_a\ v \\
[\![\mathsf{add}\ e_1\ e_2]\!]_a &= bind_a\ [\![e_1]\!]_a\ (\lambda v_1.\ bind_a\ [\![e_2]\!]_a\ (\lambda v_2.\ return_a\ (v_1 + v_2))) \\
[\![\mathsf{choose}\ e_1\ e_2]\!]_a &= merge_a\ [\![e_1]\!]_a\ [\![e_2]\!]_a
\end{aligned}
$$

Figure 2.6: blah

Because Figure 2.6 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

$$
\begin{aligned}
return_{list}\ v &= [v] \\
bind_{list}\ vs\ f &= concat\ (map\ f\ vs) \\
merge_{list}\ vs_1\ vs_2 &= append\ vs_1\ vs_2
\end{aligned}
\tag{2.23}
$$

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

$$
\begin{aligned}
[\![\mathsf{choose}\ 4\ 5]\!]_{list} &= merge_{list}\ [\![4]\!]_{list}\ [\![5]\!]_{list} \\
&= merge_{list}\ (return_{list}\ 4)\ (return_{list}\ 5) \\
&\equiv merge_{list}\ [4]\ [5] \\
&\equiv append\ [4]\ [5] \\
&\equiv [4, 5]
\end{aligned}
\tag{2.24}
$$

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

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

$$
\begin{aligned}
&\equiv \; \mathit{concat}\ [\mathit{bind}_{\mathit{list}}\ [4,5]\ (\lambda v_2.\ \mathit{return}_{\mathit{list}}\ (4+v_2)), \qquad (2.25)\\
&\qquad\qquad \mathit{bind}_{\mathit{list}}\ [4,5]\ (\lambda v_2.\ \mathit{return}_{\mathit{list}}\ (5+v_2))]\\
&\equiv \; \mathit{concat}\ [\mathit{concat}\ (\mathit{map}\ (\lambda v_2.\ \mathit{return}_{\mathit{list}}\ (4+v_2))\ [4,5]),\\
&\qquad\qquad \mathit{concat}\ (\mathit{map}\ (\lambda v_2.\ \mathit{return}_{\mathit{list}}\ (5+v_2))\ [4,5])]\\
&\equiv \; \mathit{concat}\ [\mathit{concat}\ [\mathit{return}_{\mathit{list}}\ (4+4), \mathit{return}_{\mathit{list}}\ (4+5)],\\
&\qquad\qquad \mathit{concat}\ [\mathit{return}_{\mathit{list}}\ (5+4), \mathit{return}_{\mathit{list}}\ (5+5)]]\\
&\equiv \; \mathit{concat}\ [\mathit{concat}\ [[8],[9]], \mathit{concat}\ [[9],[10]]]\\
&\equiv \; \mathit{concat}\ [[8,9],[9,10]]\\
&\equiv \; [8,9,9,10]
\end{aligned}
$$

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

The semantic function $[\![\cdot]\!]_a$ can be parameterized not just on the set and list monads, but any monad a for which 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.

As evidenced by the long derivations in (2.24) and (2.25), 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 $[\![\cdot]\!]_a$ 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 such semantics categorically in Chapter 5 and Chapter 7.

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

$$\begin{aligned}
bind_a\ (return_a\ x)\ f &\equiv f\ x && \text{left identity} \\
bind_a\ m\ return_a &\equiv m && \text{right identity} \\
bind_a\ (bind_a\ m\ f)\ g &\equiv bind_a\ m\ (\lambda x.\, bind_a\ (f\ x)\ g) && \text{associativity}
\end{aligned} \tag{2.26}$$

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 relation for the λ-calculus in which the combinators are defined. When programming in Haskell, this helpfully implies 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 differently for each category.

Lastly, there are other kinds of categories besides monads that are useful targets for categorical semantics, particularly **idioms** and **arrows**. Each has its own combinators and laws. We do not review them here because they are obscure enough that the chapters that

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

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

Figure 2.7: blah

use them necessarily review and explain them.

2.5 Abstract Interpretation

When we want to discover something about programs without actually evaluating them, it is often helpful to define an **abstract interpretation**: an evaluation method that operates on just a few 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. In either view, the properties or sets of values are thought of as **abstract values**.

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

As with concrete semantics, any of a language's abstract semantics can be specified using inference rules or semantic functions. Type systems and type checkers are typically defined by inference rules. Because Chapter 7 defines an abstract interpretation using a semantic function, we give a small example of that approach here.

Figure 2.7 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 meaning of a term is a bound on the number of nondeterministic values computed. The abstract values are the

lengths of lists or cardinalities of sets; i.e. natural numbers. For example, the abstract meaning of add (choose 4 5) (choose 4 5) is

$$\begin{aligned}
\mathcal{L}\,[\![\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ (\mathsf{choose}\ 4\ 5)]\!] &= \mathcal{L}\,[\![\mathsf{choose}\ 4\ 5]\!] \cdot \mathcal{L}\,[\![\mathsf{choose}\ 4\ 5]\!] \\
&= (\mathcal{L}\,[\![4]\!] + \mathcal{L}\,[\![5]\!]) \cdot (\mathcal{L}\,[\![4]\!] + \mathcal{L}\,[\![5]\!]) \\
&= (1+1) \cdot (1+1) \\
&= 4
\end{aligned} \tag{2.27}$$

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

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

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

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

Proof. By structural induction on e. □

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

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

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

Proof. By structural induction on e. □

This usually means that the abstraction is not very abstract, or that the concrete semantics is itself an abstraction of something else (as is the case here).

Chapter 3

Computing in Cantor's Paradise

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

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

David Hilbert

3.1 Motivation

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

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

Such a DSL should have two meanings: an exact mathematical semantics, and an approximate computational one. A traditional, denotational approach is to give the exact as a transformation to first-order set theory, and because set theory is unlike any intended implementation language, the approximate as a transformation to a lambda calculus. However,

deriving approximations while switching target languages is rife with opportunities to commit errors.

A more certain way is to define the exact semantics in a proof assistant like HOL [31] 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 [23] is devoted to formalizing early-1900s measure theory in HOL. Our work in Bayesian inference would require at least three times as much formalization, even given the work we could build on.

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

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

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

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

We define that lambda calculus, λ_{ZFC}, and a call-by-value, big-step reduction semantics.

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

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 lambda calculus [26] 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 $\lambda^{-}_{\mathrm{ZFC}}$ 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 [21] 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 "$\varnothing$" and "$\in$", and the **empty set axiom** $\forall\, x.\, x \notin \varnothing$.

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

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

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

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

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

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

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

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

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

For example, after $\{x, ...\}$ is defined, $\bigcup\{\{x, y\}, \{y, z\}\} = \{x, y, z\}$. Also, because all objects are sets, "$\bigcup$" can extract the object in a singleton set: if $A = \{x\}$, then $x = \bigcup A$.

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

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

It is not hard to show by Axiom 5 that (after $\mathbb{N}$ is defined) $\{m \mid n \in \mathbb{N} \wedge R(n, m)\}$ increments every natural number, yielding the set of positive naturals. But the syntax is cumbersome, so we define $\{F(x) \mid x \in A\} :\equiv \{y \mid x \in A \wedge y = F(x)\}$, analogous to `map F A`, for **functional replacement**. Now the more familiar $\{n + 1 \mid n \in \mathbb{N}\}$ is the positive naturals.

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

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

finds "the y such that $Q(y)$."

From the six axioms so far, we can define $A \cup B$ (binary union), $\{x, ...\}$ (literal finite sets), $\langle x, y, z, ...\rangle$ (ordered pairs and lists), $\{x \in A \mid Q(x)\}$ (bounded selection), $A \backslash B$ (relative complement), $\bigcap A$ ("big" intersection), $\bigcup_{x \in A} F(x)$ (indexed union), $A \times B$ (cartesian product), and $A \to B$ (total function spaces). For details, we recommend Paulson's remarkably lucid development in HOL [44].

3.3.1 The Gateway to Cantor's Paradise: Infinity

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

Example 3.1 (interpreting a grammar)**.** We want to interpret $z ::= \varnothing \mid \langle \varnothing, z\rangle$. It should mean the least fixpoint of a function F_z, which, given a subset of z's language, returns a larger subset. To define F_z, replace "|" with "∪", the terminal $\varnothing$ with $\{\varnothing\}$, and the rule $\langle \varnothing, z\rangle$ with functional replacement:

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

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

We follow Von Neumann, defining $0 := \varnothing$ as the **first ordinal number** and $s(n) := n \cup \{n\}$ to generate **successor ordinals**. Then $1 := s(0) = \{0\}$, $2 := s(1) = \{0, 1\}$, and $3 := s(2) = \{0, 1, 2\}$, and so on, so that every ordinal is defined as the set of its predecessors. The set of such numbers is the language of $n ::= 0 \mid s(n)$, which should be the least fixpoint of $F_n(N) := \{0\} \cup \{s(n) \mid n \in N\}$, similar to (3.2). Before we can prove this set exists, we must assume *some* fixpoint exists.

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

I is a bounding set, so it may contain more than just finite ordinals. But F_n is monotone in I, so by the Knaster-Tarksi theorem (suitably restricted [43]),

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

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

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

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

By induction, $Z(n)$ exists for every $n \in \omega$; therefore $Z(\omega)$ exists, so (3.4) is a conservative extension of set theory. It is not hard to prove (also by induction) that $Z(\omega)$ is the set of all finite lists of $\varnothing$, and that it is the least fixpoint of F_z. ♢

Similarly to building the language $Z(\omega)$ of z in (3.4), we can build the set $\mathcal{V}(\omega)$ of all hereditarily finite sets (see Axiom 3) by iterating $\mathcal{P}$ instead of F_z:

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

The set ω is not just a model of the natural numbers. It is also a number itself: the **first countable ordinal**. Indeed, ω is strikingly similar to every finite ordinal in two ways. First, it is defined as the set of its predecessors. Second, it has a successor $s(\omega) = \omega \cup \{\omega\}$. (Imagine it as $\{0, 1, 2, ..., \omega\}$.) 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):

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

It is a theorem of ZFC that every set first appears in $\mathcal{V}(\alpha)$ for some ordinal 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. $f \in 3 \to A$ is a length-3 sequence of A's elements. (An ordinal is comprised of its predecessors, so $3 = \{0, 1, 2\}$.) A **well-order** of A is a bijective sequence of A's elements.

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

Because the bijective sequence f is not unique, a well-ordering primitive could make 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 $|A| = \bigcap\{\mathit{domain}(f) \mid f \in F\}$.

3.3.3 Infinity's Infinity: An Inaccessible Cardinal

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

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

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 $\mathit{Ord}(\kappa) \wedge (\kappa > \omega) \wedge GU(\mathcal{V}(\kappa))$. □

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

Inaccessible cardinals are not usually assumed but are widely believed consistent. Set theorists regard them as no more dangerous than 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 Sect. 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 $\lambda_{\mathbf{ZFC}}$'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, appli-

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

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

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

cation, if, and domain-specific primitives, for membership, union, extraction (take), powerset, functional replacement (image), and cardinality. Values v are also typical: booleans and lambdas, and the domain-specific constants $\varnothing$ and 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 $\langle t_{\text{set}}, \varnothing\rangle$ encodes $\varnothing$.

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

3.4.1 An Infinite Set Rule For Finite BNF Grammars

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

To define λ_{ZFC}'s terms, we first extend BNF with a set rule: $\{y^{*\alpha}\}$, where α is a cardinal 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

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

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

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

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 $\mathcal{V}(\kappa)$. ♢

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

Theorem 3.5. *Let F be a unary function. Define V by transfinite recursion:*

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

*Let 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 $\bigcup$. □

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

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

Theorem 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. □

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. ♢

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. ♢

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. ♢

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 $\varnothing$ 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*.

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

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

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

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

To use Theorem 3.6, we need to iterate a single function. Note that the language pair $\langle E, V\rangle = \langle\{e, ...\}, \{v, ...\}\rangle$ is isomorphic to the single set of tagged terms $EV = \{\langle 0, e\rangle, ..., \langle 1, v\rangle, ...\}$. Binary **disjoint union**, denoted $E \sqcup V$, creates such sets. We define F_{ev} by $F_{ev}(E \sqcup V) = F_e(E,V) \sqcup F_v(E,V)$. By Theorem 3.6, its least fixpoint is F_{ev}^{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, $\widehat{C}(v_A) := set(|snd(v_A)|)$ extracts the tagged set from v_A, applies $|\cdot|$, and recursively tags the

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

(a) Standard call-by-value reduction rules

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

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

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

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

(b) 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

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

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

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

Figure 3.4 shows the reduction rules that define the reduction relation "$\Downarrow$". Figure 3.4a has standard call-by-value rules: values reduce to themselves, and applications reduce by substitution. Figure 3.4b has the 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:

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

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

For $F_{\Downarrow}$ itself to be nonmonotone, for some $R \subseteq R' \subseteq \mathcal{U}$, there would have to be a conclusion $c \in F_{\Downarrow}(R)$ that is not in $F_{\Downarrow}(R')$. In other words, having more known conclusions could falsify a premise. None of the rules in Fig. 3.4 can do so.

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

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

Theorem 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 $v ::= \langle t_{\text{set}}, \{v^{*\kappa}\}\rangle$. □

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, $\lambda \mathsf{x}.\, \mathsf{x} = \langle t_\lambda, \langle t_{\text{var}}, 0\rangle\rangle = \lambda \mathsf{y}.\, \mathsf{y}$.

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

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

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

To do anything useful, we need a small set library. The definitions are similar to the metalanguage definitions we omitted in Section 3.3, and we similarly elide most of the λ_{ZFC} definitions. However, some deserve special mention.

Because λ_{ZFC} has only *functional* replacement, we cannot define unbounded "∀" and

"$\exists$". But we can define bounded quantifiers in terms of bounded selection, or

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

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

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

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

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

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

defines ordered pairs, then

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

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

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

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

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

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

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

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

Every $\mathsf{f} \in \mathsf{A} \to \mathsf{B}$ is shaped $\mathsf{f} = \{\langle \mathsf{x}_1, \mathsf{y}_1 \rangle, \langle \mathsf{x}_2, \mathsf{y}_2 \rangle, ...\}$ and is total on A. To distinguish these hash tables from lambdas, we call them **mappings**. Mappings can be applied by $\mathsf{ap\ f\ x} := \iota\, \mathsf{y} \in (\mathsf{range\ f}).\, \langle \mathsf{x}, \mathsf{y} \rangle \in \mathsf{f}$, but we write just $\mathsf{f\ x}$. We define

$$e_f|_{e_A} :\equiv\ \mathsf{image}\ (\lambda \mathsf{x}.\, \langle \mathsf{x}, e_f\ \mathsf{x} \rangle)\ e_A \tag{3.19}$$

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

$$\lambda\, x \in e_A.\, e_y\ \ :\equiv\ \ (\lambda\, x.\, e_y)|_{e_A} \tag{3.20}$$

to define mappings. (XXX: this is not what restriction is in the preimage paper)

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

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

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

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

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

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

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

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

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

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

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

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

Theorem 3.13. *Suppose* $=_{\mathsf{X}}$ *is an equivalence relation on* X*, and define its quotient* $\mathbb{X} := \mathsf{quotient}\ \mathsf{X}\ (=_{\mathsf{X}})$*. If* op *is surjective on* X *and a binary morphism for* $=_{\mathsf{X}}$*, then* $(\mathsf{lift2}_{\mathsf{set}}\ \mathsf{op}\ \mathsf{A}\ \mathsf{B}) \in \mathbb{X}$ *for all* $\mathsf{A} \in \mathbb{X}$ *and* $\mathsf{B} \in \mathbb{X}$*.*

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

Now for real limits. If $\mathbb{R}^+$, '$<$', and $|\cdot|$ are defined, we can define $\mathsf{conv\text{-}zero?}_{\mathbb{R}}$, which is like (3.21) but operates on real sequences $\mathsf{xs} \in \omega \to \mathbb{R}$. We then define $\mathsf{limit}_{\mathbb{R}}\ \mathsf{xs} := \iota\, \mathsf{y} \in \mathbb{R}.\, \mathsf{conv\text{-}zero?}_{\mathbb{R}}\ (\mathsf{map}\ (-\ \mathsf{y})\ \mathsf{xs})$ to calculate their limits.

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

3.8 Example: Computable Real Limits

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

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

The proof obligations are related to topological theorems [38] 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}. $\mathsf{A} \Rightarrow \mathsf{B}$ is a lambda or mapping type. $\mathsf{A} \to \mathsf{B}$ is the set of total mappings from A to B. A set is a membership proposition.

3.8.1 The Limit Monad

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

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

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

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

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

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

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

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

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

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

3.8.2 The Computable Limit Monad

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

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

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

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

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

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

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

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

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

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

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

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

So if flip xss converges uniformly, then

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

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

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

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

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

We have a product topology, so for the first obligation, pointwise continuity is enough. Let g := 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]. ♢

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

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

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

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

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

Implementation. We have transliterated $\mathsf{return}'_{\mathsf{lim}}$, $\mathsf{bind}'_{\mathsf{lim}}$, $\mathsf{exp}'_{\mathsf{lim}}$, $\mathsf{arctan}'_{\mathsf{lim}}$ and π'_{lim} into Racket [17], using its built-in models of ω and $\mathbb{Q}$. Even without optimizations, π'_{lim} 141 yields a rational approximation in a few milliseconds that is correct to 200 digits. More importantly, $\mathsf{exp}'_{\mathsf{lim}}$, $\mathsf{arctan}'_{\mathsf{lim}}$ and π'_{lim} are almost identical to their counterparts in the uncomputable limit monad, and meet their proof obligations. The code is clean, short, correct and reasonably fast, and resides in a directory named `flops2012` at `https://github.com/ntoronto/plt-stuff/`.

3.9 Related Work

O'Connor's completion monad [40] 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 [31], CIC [8], MT [7] (Map Theory) and EFL* [16] continue Church's programme to found mathematics on the lambda calculus. Like λ_{ZFC}, interpreting them in set theory seems to require a slightly stronger theory than plain ZFC. HOL and CIC ensure consistency using types, and use the Curry-Howard correspondence to extract programs.

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

The key difference between λ_{ZFC} and these higher-order logics is that λ_{ZFC} is not a logic. It is a programming language with infinite terms, which by design includes a transitive model of set theory (Theorem 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 highly nontrivial. Applying a ZFC theorem in HOL or CIC requires re-proving it to the satisfaction of a type checker.

The infinitary lambda calculus [26] 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 [41] describes many Turing machine extensions, including completion of transfinite computations. Much of the research is for discovering the properties of computation in physically plausible extensions. While λ_{ZFC} might offer a civilized way to program such machines, we do not think of our work as hypercomputation, but as approaching computability from above.

3.10 Conclusions and Future Work

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

Our main future work is using λ_{ZFC} to define languages for Bayesian inference, then deriving implementations that compute converging probabilities. Overall, we no longer have to hold back when a set-theoretic construction could be defined elegantly with untyped lambdas or recursion, or generalized precisely with higher-order functions. If we can derive a computable replacement, we might help someone in Cantor's Paradise compute the apparently uncomputable.

Chapter 4

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

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

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

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 lambda-calculus. For example, a function that does not terminate on any input because of runaway recursion (i.e. an infinite loop) is

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

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

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

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

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

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

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.
- $\mathsf{x} \Rightarrow \mathsf{y}$ denotes a partial function.

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

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

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

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 values that cannot are those that *must* be defined **nonconstructively**: by proving existence and uniqueness, without giving a bound (no matter how loose) on the length or cardinality of the value. Mathematicians avoid such nonconstructive definitions, and most would consider that definition of "nonconstructive" too liberal.[1]

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* $\mathsf{A} : \mathsf{Set}\ \mathsf{x}$ *and* $\mathsf{B} : \mathsf{Set}\ \mathsf{x}$*,* $\mathsf{A} = \mathsf{B}$ *if and only if* $\mathsf{A} \subseteq \mathsf{B}$ *and* $\mathsf{B} \subseteq \mathsf{A}$*.*

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

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

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

[3] Assuming the existence of an inaccessible cardinal, which is a modest assumption, as ZFC+κ is a smaller theory than Coq's [6].

```
domain : (X ⇀ Y) ⇒ Set X
domain := image fst

range : (X ⇀ Y) ⇒ Set Y
range := image snd

preimage : (X ⇀ Y) ⇒ Set Y ⇒ Set X
preimage g B := {a ∈ domain g | g a ∈ B}

restrict : (X ⇀ Y) ⇒ Set X ⇒ (X ⇀ Y)
restrict g A := λa ∈ (A ∩ domain g). g a
```

Figure 4.1: λ_{ZFC} code for common operations on partial mappings.

4.4 Internal Equality and External Equivalence

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

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

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

Definition 4.2 (observational equivalence)**.** *For terms e_1 and e_2, $e_1 \equiv e_2$ when $e_1 = e_2$, or both e_1 and e_2 do not terminate.*

It might seem helpful to define basic equivalence even more coarsely, so that we can say $\lambda\,\mathsf{a}.\,2$ is equivalent to $\lambda\,\mathsf{a}.\,1+1$. However, we want internal equality and basic external equivalence to be similar, and we want to be able to extend "$\equiv$" with type-specific rules.

4.5 Additional Functions and Syntactic Forms

We use heavily sugared syntax, with automatic currying (including primitive applications, so image fst means $\lambda\,\mathsf{A}.$ image fst A), binding forms such as indexed unions $\bigcup_{x \in e_A} e$, destructuring binds as in swap $\langle \mathsf{a}, \mathsf{b} \rangle := \langle \mathsf{b}, \mathsf{a} \rangle$, and comprehensions like $\{\mathsf{a} \in \mathsf{A} \mid \mathsf{a} \in \mathsf{B}\}$. We assume logical

operators, bounded quantifiers, and typical set operations are defined. To refer to binary operators as values, we enclose them in parentheses, as in $(\in)$ and $(\subseteq)$.

A less typical set operation we use is disjoint union:

$$\begin{aligned} &(\uplus) : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \\ &\mathsf{A} \uplus \mathsf{B} \ :=\ \mathsf{if}\ (\mathsf{A} \cap \mathsf{B} = \varnothing)\ (\mathsf{A} \cup \mathsf{B})\ (\mathsf{take}\ \varnothing) \end{aligned} \tag{4.4}$$

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

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

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

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 $\{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, ...\}$. (It is fine to think of such encodings as infinite hash tables.) We call these encodings **mappings**. We use **function** to mean either a lambda or a mapping, and use adjacency (e.g. $(\mathsf{f}\ \mathsf{x})$) to apply either kind.

Syntax for constructing unnamed mappings is defined by

$$\lambda\, x_a \in e_A.\, e_b \ :=\ \mathsf{mapping}\ (\lambda\, x_a.\, e_b)\ e_A \tag{4.5}$$

$$\begin{aligned} &\mathsf{mapping} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \\ &\mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \ :=\ \mathsf{image}\ (\lambda\, \mathsf{a}.\, \langle \mathsf{a}, \mathsf{f}\ \mathsf{a}\rangle)\ \mathsf{A} \end{aligned} \tag{4.6}$$

For symmetry with partial functions $\mathsf{x} \Rightarrow \mathsf{y}$, mapping returns a member of the set $\mathsf{X} \rightharpoonup \mathsf{Y}$ of all *partial* mappings from X to Y. Figure 4.1 defines other common mapping operations: domain, range, preimage (which finds the mapping inputs that would produce outputs in a

given set) and restrict (which narrows the domain of a mapping).

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

$$\begin{aligned} &\pi : \mathsf{J} \Rightarrow (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{X} \\ &\pi\ \mathsf{j}\ \mathsf{f}\ :=\ \mathsf{f}\ \mathsf{j} \end{aligned} \tag{4.7}$$

produces projections. This is particularly useful when f is unnamed or unknown: $\pi\ \mathsf{j}$ takes any such f and returns the element at index j.

Chapter 5

Semantics of Countable Models

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

Bayesian practitioners define *models*, or probabilistic relationships among objects of study, without regard to whether future calculations are closed-form or tractable. They are loath to make simplifying assumptions. (If some probabilistic phenomenon is best described by an unsolvable integral or infinitely many distributions, so be it.) When they must approximate, they often create two models: an "ideal" model first, and a second model that approximates it.

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

automatable part.

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

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

5.1.1 Theory of Probability

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 model. 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.

5.1.2 Approach

For three categories of Bayesian notation, we

1. Manually interpret an unambiguous subclass of typical notation.
2. Mechanize the interpretation with a semantic function.
3. If necessary, create an approximation and prove convergence.
4. Implement the approximation in Racket [17].

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 three 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 $\mathbf{D}[\![\cdot]\!]$.

Each semantic function transforms its syntactic category into λ_{ZFC}, in which we write all of our mathematics. 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

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

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. $\mathsf{E} : \Omega \to \{\mathsf{even}, \mathsf{odd}\}$ must simulate the universe until the die is still, and then recognize the outcome. Hopefully, the probability that $\mathsf{E}\ \omega = \mathsf{even}$ is $\frac{1}{2}$.

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

$\frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$.

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

Random variables enable a kind of probabilistic abstraction. The example does it twice. The first makes calculating the probability that $\mathsf{E}\ \omega = \mathsf{even}$ tractable. The second is an optimization. In fact, redefining 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 nondeterminism from assigning probabilities. It allows us to interpret expressions involving random variables without considering probabilities at all.

5.2.2 Interpreting Random Variable Expressions As Computations

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

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

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

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

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.

We abstract lifting and application with the combinators

$$
\begin{aligned}
\mathsf{pure_{rv}\ c} &= \lambda\omega.\, \mathsf{c} \\
\mathsf{ap^*_{rv}\ F\ X_1\ ...\ X_n} &= \lambda\omega.\, ((\mathsf{F}\ \omega)\ (\mathsf{X_1}\ \omega)\ ...\ (\mathsf{X_n}\ \omega))
\end{aligned}
\tag{5.3}
$$

so that $(+\ 4\ \mathsf{A})$ means $\mathsf{ap^*_{rv}}\ (\mathsf{pure_{rv}}\ +)\ (\mathsf{pure_{rv}}\ 4)\ \mathsf{A} = \cdots = \lambda\omega.\, (+\ 4\ (\mathsf{A}\ \omega))$. These combinators define an **idiom** [36], 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 $(\mathsf{I\ a}) = (\Omega \Rightarrow \mathsf{a})$ for some Ω.

$\mathcal{R}[\![\cdot]\!]$ (Fig. 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. Fig. 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 converge when applied. These lambdas do not represent random variables, which are total. We will be able to recover mappings by restricting them, as in $\mathcal{R}[\![(+\ 4\ \mathsf{A})]\!]|_\Omega$.

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

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

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

5.2.3 Implementation in Racket

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

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

5.3 The Query Language

It is best to regard statements in Bayesian notation 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** $\mathbb{P} : \mathcal{P}(\Omega) \to [0, 1]$. Any discrete probability measure is uniquely determined by its value on singleton sets, or by a **probability mass function** $\mathsf{P} : \Omega \to [0, 1]$. It is easy to convert P to a probability measure:

$$\mathsf{sum}\ \mathsf{P}\ \mathsf{A} = \sum_{\omega \in \mathsf{A}} \mathsf{P}\ \omega \tag{5.4}$$

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

A **discrete probability space** $\langle \Omega, \mathsf{P} \rangle$ embodies all probabilistic nondeterminism introduced by 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 $\langle \Omega, \mathsf{P} \rangle$. For example, suppose we want to calculate, as in Example 5.1, the probability of an even die outcome. We must apply $\mathbb{P}$ to the correct subset of Ω. Suppose that $\Omega = \{1, 2, 3, 4, 5, 6\}$ and that $\mathsf{P} = [1, 2, 3, 4, 5, 6 \to \frac{1}{6}]$ determines $\mathbb{P}$. The probability that E outputs even is

$$\mathbb{P}\ \{\omega \in \Omega \mid \mathsf{E}\ \omega = \mathsf{even}\} \ = \ \mathbb{P}\ \{2, 4, 6\} \ = \ \mathsf{sum}\ \mathsf{P}\ \{2, 4, 6\} \ = \ \tfrac{1}{2} \tag{5.5}$$

This is a **probability query**.

Alternatively, we could use a **distribution query** to calculate E's distribution $\mathbb{P}_{\mathsf{E}}$, and then apply it to $\{\mathsf{even}\}$. Measure-theoretic probability elegantly defines $\mathbb{P}_{\mathsf{E}}$ as $\mathbb{P} \circ \mathsf{E}^{-1}$, but for now we do not need a measure. We only need the probability mass function $\mathsf{P}_{\mathsf{E}}\ \mathsf{e} = \mathsf{sum}\ \mathsf{P}\ (\mathsf{E}^{-1}\ \{\mathsf{e}\})$. Applying it yields

$$\mathsf{P}_{\mathsf{E}}\ \mathsf{even}\ =\ \mathsf{sum}\ \mathsf{P}\ \left(\mathsf{E}^{-1}\ \{\mathsf{even}\}\right)\ =\ \mathsf{sum}\ \mathsf{P}\ \{2,4,6\}\ =\ \tfrac{1}{2} \tag{5.6}$$

More abstractly, we can calculate discrete distribution queries using

$$\mathsf{dist}\ \mathsf{X}\ \langle \Omega, \mathsf{P}\rangle\ =\ \lambda \mathsf{x} \in \mathsf{S}_{\mathsf{X}}.\, \mathsf{sum}\ \mathsf{P}\ \left(\mathsf{X}|_{\Omega}^{-1}\ \{\mathsf{x}\}\right) \tag{5.7}$$

where $\mathsf{S}_{\mathsf{X}} = \mathsf{image}\ \mathsf{X}\ \Omega$. Recall that $\mathsf{X}|_{\Omega}$ 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 $\mathsf{P}[\cdot]$ replaces applying $\mathbb{P}$ and sets become propositions. For example, a common way to write "the probability of an even die outcome" in practice is $\mathsf{P}[\mathsf{E} = \mathsf{even}]$.

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 $\{\mathit{true}\}$. For $\mathsf{E} = \mathsf{even}$, for example, it is $\mathcal{R}[\![\mathsf{E} = \mathsf{even}]\!]|_{\Omega}^{-1}\ \{\mathit{true}\}$. In general,

$$\mathsf{sum}\ \mathsf{P}\ \left(\mathcal{R}[\![\mathsf{e}]\!]|_{\Omega}^{-1}\ \{\mathit{true}\}\right)\ =\ \mathsf{dist}\ \mathcal{R}[\![\mathsf{e}]\!]\ \langle \Omega, \mathsf{P}\rangle\ \mathit{true} \tag{5.8}$$

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

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 $\mathsf{P}[\mathsf{E} = \mathsf{e}]$ 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 $\mathbf{D}[\![\mathsf{E}]\!]$ 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, sum P A is an infinite series. With A as 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 $\langle \omega_1, \omega_2, ... \rangle$ is an enumeration of Ω. Let $\mathsf{z} \in \mathbb{N}$ be the length of the prefix $\Omega_\mathsf{z} = \{\omega_1, ..., \omega_\mathsf{z}\}$ and let $\mathsf{P}_\mathsf{z}\ \omega = (\mathsf{P}\ \omega)/\,(\mathsf{sum}\ \mathsf{P}\ \Omega_\mathsf{z})$. Then P_z converges to P. We define $\mathsf{finitize}\ \langle \Omega, \mathsf{P} \rangle = \langle \Omega_\mathsf{z}, \mathsf{P}_\mathsf{z} \rangle$ with $\mathsf{z} \in \mathbb{N}$ as a free variable.

5.3.5 Implementation in Racket

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

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

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

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

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

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

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 `finitize`d 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 language without conditional queries is of little more use to them than a general-purpose language.

Measure-theoretic conditional probability is too involved to accurately summarize here.

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

$$\mathsf{P}[\mathsf{A} \mid \mathsf{B}] \;=\; (\mathbb{P}\ \mathsf{A} \cap \mathsf{B}) / (\mathbb{P}\ \mathsf{B}) \tag{5.9}$$

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

Example 5.2. Extend Example 5.1 with random variable L $\omega = \mathsf{low}$ if $\omega \leq 3$, else high. The probability that $\mathsf{E} = \mathsf{even}$ given $\mathsf{L} = \mathsf{low}$ is

$$\mathsf{P}[\mathsf{E} = \mathsf{even} \mid \mathsf{L} = \mathsf{low}] \;=\; \frac{\mathsf{P}[\mathsf{E} = \mathsf{even} \wedge \mathsf{L} = \mathsf{low}]}{\mathsf{P}[\mathsf{L} = \mathsf{low}]} \;=\; \frac{\sum_{\omega \in \{2\}} \mathsf{P}\ \omega}{\sum_{\omega \in \{1,2,3\}} \mathsf{P}\ \omega} \;=\; \frac{\frac{1}{6}}{\frac{1}{2}} \;=\; \frac{1}{3} \tag{5.10}$$

Less precisely, there are proportionally fewer even outcomes when $\mathsf{L} = \mathsf{low}$. ♢

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

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

$$\begin{aligned} \mathsf{cond}\ \mathsf{Y}\ \mathsf{y}\ \langle \Omega, \mathsf{P} \rangle \;=\; \langle \Omega', \mathsf{P}' \rangle \ \text{ where } \ \Omega' &= \mathsf{Y}|_{\Omega}^{-1}\ \{\mathsf{y}\} \\ \mathsf{P}' &= \lambda \omega \in \Omega'.\, (\mathsf{P}\ \omega) / (\mathsf{sum}\ \mathsf{P}\ \Omega') \end{aligned} \tag{5.11}$$

Then $\mathsf{P}[\mathsf{E} = \mathsf{even} \mid \mathsf{L} = \mathsf{low}]$ means $\mathsf{dist}\ \mathsf{E}\ (\mathsf{cond}\ \mathsf{L}\ \mathsf{low}\ \langle \Omega, \mathsf{P} \rangle)\ \mathsf{even}$.

We approximate cond by applying $\mathsf{finitize}$ to the probability space. 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

$$\mathsf{t_i} \ ::= \ \mathsf{X_i} \sim \mathsf{e_i};\ \mathsf{t_{i+1}} \mid \mathsf{X_i} := \mathsf{e_i};\ \mathsf{t_{i+1}} \mid \mathsf{e_a} = \mathsf{e_b};\ \mathsf{t_{i+1}} \mid \epsilon \tag{5.12}$$

Further, they can always be made ***well-formed***: an $\mathsf{e_j}$ may refer to some $\mathsf{X_i}$ only when $\mathsf{j} > \mathsf{i}$ (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 $\mathsf{P_L} = [\mathsf{low}, \mathsf{high} \mapsto \frac{1}{2}]$, then E's distribution depends on L's output.

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

$$\mathsf{L} \sim \mathsf{P_L};\ \mathsf{E} \sim \left(\mathsf{P_{E|L}}\ \mathsf{L}\right) \tag{5.13}$$

If L is a measure-theoretic random variable, $\left(\mathsf{P_{E|L}}\ \mathsf{L}\right)$ does not type-check: $\mathsf{L} : \Omega \to \mathsf{S_L}$ is clearly not in $\mathsf{S_L}$. The *intent* is that E's distribution depends on L, and that $\mathsf{P_{E|L}}$ specifies how. ♢

We can regard $\mathsf{L} \sim \mathsf{P_L}$ as a constraint: for every model $\langle \Omega, \mathsf{P}, \mathsf{L} \rangle$, $\mathsf{dist}\ \mathsf{L}\ \langle \Omega, \mathsf{P} \rangle$ must be $\mathsf{P_L}$. Similarly, $\mathsf{E} \sim \left(\mathsf{P_{E|L}}\ \mathsf{L}\right)$ means E's conditional distribution is $\mathsf{P_{E|L}}$. We have been using

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

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

the model $\Omega = \{1,2,3,4,5,6\}$, $\mathsf{P} = [1,2,3,4,5,6 \mapsto \frac{1}{6}]$, and the obvious E and L. It is not hard to verify that this is also a model:

$$\begin{aligned} &\Omega = \{\mathsf{low},\mathsf{high}\} \times \{\mathsf{even},\mathsf{odd}\} \qquad \mathsf{L}\ \omega = \omega_1 \qquad \mathsf{E}\ \omega = \omega_2 \\ &\mathsf{P} = [\langle \mathsf{low},\mathsf{even}\rangle, \langle \mathsf{high},\mathsf{odd}\rangle \mapsto \tfrac{1}{6}, \langle \mathsf{low},\mathsf{odd}\rangle, \langle \mathsf{high},\mathsf{even}\rangle \mapsto \tfrac{2}{6}] \end{aligned} \tag{5.14}$$

The construction of Ω, L and E in (5.14) 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:

$$\begin{aligned} \mathsf{P}\ \omega &= (\mathsf{P}_{\mathsf{L}}\ (\mathsf{L}\ \omega)) \times \big(\mathsf{P}_{\mathsf{E}|\mathsf{L}}\ (\mathsf{L}\ \omega)\ (\mathsf{E}\ \omega)\big) \\ \text{or} \quad \mathsf{P} &= \mathcal{R}[\![(\mathsf{P}_{\mathsf{L}}\ \mathsf{L}) \times \big((\mathsf{P}_{\mathsf{E}|\mathsf{L}}\ \mathsf{L})\ \mathsf{E}\big)]\!] \end{aligned} \tag{5.15}$$

If $\mathsf{K}_{\mathsf{L}} = \mathcal{R}[\![\mathsf{P}_{\mathsf{L}}]\!]$ and $\mathsf{K}_{\mathsf{E}} = \mathcal{R}[\![\big(\mathsf{P}_{\mathsf{E}|\mathsf{L}}\ \mathsf{L}\big)]\!]$—which interpret (5.13)'s statements' right-hand sides—then $\mathsf{P} = \mathcal{R}[\![(\mathsf{K}_{\mathsf{L}}\ \mathsf{L}) \times (\mathsf{K}_{\mathsf{E}}\ \mathsf{E})]\!]$. This can be generalized.

Definition 5.4 (discrete product model)**.** *Given a well-formed, discrete conditional theory* $\mathsf{X}_1 \sim \mathsf{e}_1; ...; \mathsf{X}_{\mathsf{n}} \sim \mathsf{e}_{\mathsf{n}}$*, let* $\mathsf{K}_{\mathsf{i}} : \Omega \Rightarrow \mathsf{S}_{\mathsf{i}} \to [0,1]$*,* $\mathsf{K}_{\mathsf{i}} = \mathcal{R}[\![\mathsf{e}_{\mathsf{i}}]\!]$ *for each* $1 \le \mathsf{i} \le \mathsf{n}$*. The* ***discrete product model*** *of the theory is*

$$\Omega = \bigtimes_{\mathsf{i}=1}^{\mathsf{n}} \mathsf{S}_{\mathsf{i}} \qquad \mathsf{X}_{\mathsf{i}}\ \omega = \omega_{\mathsf{i}}\ (1 \le \mathsf{i} \le \mathsf{n}) \qquad \mathsf{P} = \mathcal{R}\left[\!\!\left[\prod_{\mathsf{i}=1}^{\mathsf{n}} (\mathsf{K}_{\mathsf{i}}\ \mathsf{X}_{\mathsf{i}}) \right]\!\!\right] \tag{5.16}$$

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** $\mathcal{R}[\![\mathsf{e}_{\mathsf{i}}]\!] : \Omega \Rightarrow \mathsf{S}_{\mathsf{i}} \to [0,1]$. In measure theory, transition kernels are used to build **product spaces** such as $\langle \Omega, \mathsf{P}\rangle$. Thus, $\mathcal{R}[\![\cdot]\!]$ links Bayesian practice to measure theory and represents an increase in expressive power in specifying distributions, by turning properly typed λ_{ZFC} terms into precisely what measure

$$\mathsf{dist_{ps}}\ \mathsf{X}\ \langle \Omega, \mathsf{P}\rangle = \langle \Omega, \mathsf{P}, \mathsf{P_X}\rangle \ \text{ where } \ \mathsf{S_X} = \mathsf{image}\ \mathsf{X}\ \Omega$$
$$\mathsf{P_X} = \lambda \mathsf{x} \in \mathsf{S_X}.\, \mathsf{sum}\ \mathsf{P}\ \left(\mathsf{X}|_{\Omega}^{-1}\ \{\mathsf{x}\}\right)$$
$$\mathsf{cond_{ps}}\ \mathsf{Y}\ \mathsf{y}\ \langle \Omega, \mathsf{P}\rangle = \langle \Omega', \mathsf{P}', \_\rangle \ \text{ where } \ \Omega' = \mathsf{Y}|_{\Omega}^{-1}\ \{\mathsf{y}\}$$
$$\mathsf{P}' = \lambda \omega \in \Omega'.\, (\mathsf{P}\ \omega)/\,(\mathsf{sum}\ \mathsf{P}\ \Omega')$$
$$\mathsf{extend_{ps}}\ \mathsf{K_i}\ \langle \Omega_{\mathsf{i}-1}, \mathsf{P_{i-1}}\rangle = \langle \Omega_{\mathsf{i}}, \mathsf{P_i}, \mathsf{X_i}\rangle$$
$$\text{where } \ \mathsf{S'_i}\ \omega = \mathsf{domain}\ (\mathsf{K_i}\ \omega), \quad \Omega_{\mathsf{i}} = (\omega \in \Omega_{\mathsf{i}-1}) \times (\mathsf{S'_i}\ \omega)$$
$$\mathsf{X_i}\ \omega = \omega_{\mathsf{j}}\ \ (\text{where } \mathsf{j} = \text{length of any } \omega \in \Omega_{\mathsf{i}-1}), \quad \mathsf{P_i} = \mathcal{R}[\![\mathsf{P_{i-1}} \times (\mathsf{K_i}\ \mathsf{X_i})]\!]$$
$$\mathsf{run_{ps}}\ \mathsf{m} = \mathsf{x} \ \text{ where } \ \langle \Omega, \mathsf{P}, \mathsf{x}\rangle = \mathsf{m}\ \langle \{\langle\rangle\}, \lambda \omega.\, 1\rangle$$

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

theory requires.

5.5.2 Interpreting Statements as Monadic Computations

Some conditional theories state more than just distributions [35, 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.

We assume the state monad's $\mathsf{return_s}$ and $\mathsf{bind_s}$. Fig. 5.4 shows the additional $\mathsf{dist_{ps}}$, $\mathsf{cond_{ps}}$ and $\mathsf{extend_{ps}}$. The first two simply reimplement dist and cond. But $\mathsf{extend_{ps}}$, which interprets statements, needs more explanation.

According to (5.16), interpreting $\mathsf{X_i} \sim \mathsf{e_i}$ results in $\Omega_{\mathsf{i}} = \Omega_{\mathsf{i}-1} \times \mathsf{S_i}$, with $\mathsf{S_i}$ extracted from $\mathsf{K_i} : \Omega_{\mathsf{i}-1} \Rightarrow \mathsf{S_i} \to [0, 1]$. A more precise type for $\mathsf{K_i}$ is the dependent type $(\omega : \Omega_{\mathsf{i}-1}) \Rightarrow (\mathsf{S'_i}\ \omega) \to [0, 1]$, which reveals a complication. To extract $\mathsf{S_i}$, we first must extract the random variable $\mathsf{S'_i} : \Omega_{\mathsf{i}-1} \to \mathcal{P}(\mathsf{S_i})$. So let $\mathsf{S'_i}\ \omega = \mathsf{domain}\ (\mathsf{K_i}\ \omega)$; then $\mathsf{S_i} = \bigcup (\mathsf{image}\ \mathsf{S'_i}\ \Omega_{\mathsf{i}-1})$.

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*

$$\mathcal{M}[\![\mathsf{X_i} := \mathsf{e_i};\ \mathsf{t_{i+1}}]\!] = \mathsf{bind_s}\ (\mathsf{return_s}\ \mathcal{R}[\![\mathsf{e_i}]\!])\ \lambda \mathsf{X_i}.\, \mathcal{M}[\![\mathsf{t_{i+1}}]\!]$$
$$\mathcal{M}[\![\mathsf{X_i} \sim \mathsf{e_i};\ \mathsf{t_{i+1}}]\!] = \mathsf{bind_s}\ (\mathsf{extend_{ps}}\ \mathcal{R}[\![\mathsf{e_i}]\!])\ \lambda \mathsf{X_i}.\, \mathcal{M}[\![\mathsf{t_{i+1}}]\!]$$
$$\mathcal{M}[\![\mathsf{e_a} = \mathsf{e_b};\ \mathsf{t_{i+1}}]\!] = \mathsf{bind_s}\ (\mathsf{cond_{ps}}\ \mathcal{R}[\![\mathsf{e_a}]\!]\ \mathcal{R}[\![\mathsf{e_b}]\!])\ \lambda \_.\, \mathcal{M}[\![\mathsf{t_{i+1}}]\!]$$
$$\mathcal{M}[\![\epsilon]\!] = \mathsf{return_s}\ \langle \mathsf{X}_1, ..., \mathsf{X_n} \rangle$$

$$\mathbf{D}[\![\mathsf{e}]\!]\ \mathsf{m} = \mathsf{run_{ps}}\,(\mathsf{bind_s}\ \mathsf{m}\ \lambda \langle \mathsf{X}_1, ..., \mathsf{X_n} \rangle.\, \mathsf{dist_{ps}}\ \mathcal{R}[\![\mathsf{e}]\!])$$
$$\mathbf{D}[\![\mathsf{e_X} \mid \mathsf{e_Y}]\!]\ \mathsf{m} = \lambda \mathsf{y}.\, \mathbf{D}[\![\mathsf{e_X}]\!]\ (\mathsf{bind_s}\ \mathsf{m}\ \lambda \langle \mathsf{X}_1, ..., \mathsf{X_n} \rangle.\, \mathcal{M}[\![\mathsf{e_Y} = \mathsf{y}]\!])$$
$$\mathbf{P}[\![\mathsf{e}]\!]\ \mathsf{m} = \mathbf{D}[\![\mathsf{e}]\!]\ \mathsf{m}\ \mathit{true}, \quad \mathbf{P}[\![\mathsf{e_A} \mid \mathsf{e_B}]\!]\ \mathsf{m} = \mathbf{D}[\![\mathsf{e_A} \mid \mathsf{e_B}]\!]\ \mathsf{m}\ \mathit{true}\ \mathit{true}$$

Figure 5.5: The conditional theory and query semantic functions.

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

Dependent cartesian products are elegantly expressed using the set monad:

$$\mathsf{return_v}\ \mathsf{x} = \{\mathsf{x}\} \qquad \mathsf{bind_v}\ \mathsf{m}\ \mathsf{f} = \bigcup (\mathsf{image}\ \mathsf{f}\ \mathsf{m}) \tag{5.17}$$

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

Fig. 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.)

$\mathbf{D}[\![\mathsf{e}]\!]$ expands to a distribution-valued computation and runs it with the ***empty probability space*** $\langle \Omega_0, \mathsf{P}_0 \rangle = \langle \{\langle\rangle\}, \lambda \omega.\, 1 \rangle$. $\mathbf{D}[\![\mathsf{e_X} \mid \mathsf{e_Y}]\!]$ conditions the probability space and hands off to $\mathbf{D}[\![\mathsf{e_X}]\!]$. $\mathbf{P}[\![\cdot]\!]$ is defined in terms of $\mathbf{D}[\![\cdot]\!]$.

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

5.5.3 Approximating Models and Queries

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

5.5.4 Implementation in Racket

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

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

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

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

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

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

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 [39].

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 0.523919127555099524791984 3—25 decimal digits—with custom MATLAB code. At `appx-z` $\geq$ `321`, our solution computes the same digits. (Though it appends the digits 9..., so Nahin should have rounded up!) Implementing it took about five minutes. But the problem is not Bayesian.

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

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

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

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

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

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

The most direct theory has infinitely many statements, one for each possible shot. Supporting such theories is future work.

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 $\mathsf{t}_1; ...; \mathsf{t}_n$ 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* $\mathcal{M}[\![\mathsf{t}_1; ...; \mathsf{t}_i; \mathsf{t}_{i+1}; ...; \mathsf{t}_n]\!]\ \langle \Omega_0, \mathsf{P}_0 \rangle = \mathcal{M}[\![\mathsf{t}_1; ...; \mathsf{t}_{i+1}; \mathsf{t}_i; ...; \mathsf{t}_n]\!]\ \langle \Omega_0, \mathsf{P}_0 \rangle$.

This notion of commutativity is too strong: distribution statements would never commute with each other. We need a weaker test than equality.

Definition 5.7 (equivalence in distribution)**.** *Suppose* $\mathsf{X}_1, ..., \mathsf{X}_k$ *are defined in* $\mathsf{t}_1, ..., \mathsf{t}_n$*. Let* $\mathsf{m} = \mathcal{M}[\![\mathsf{t}_1, ..., \mathsf{t}_n]\!]$*, and* m' *be a (usually different) probability space monad computation. We write* $\mathsf{m} \equiv_{\mathbf{D}} \mathsf{m}'$ *and call* m *and* m' ***equivalent in distribution*** *when* $\mathbf{D}[\![\mathsf{X}_1, ..., \mathsf{X}_k]\!]\ \mathsf{m} = \mathbf{D}[\![\mathsf{X}_1, ..., \mathsf{X}_k]\!]\ \mathsf{m}'$.

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

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

Definition 5.9 (commutativity in distribution)**.** *We say* t_i *and* t_{i+1} *commute* ***in distribution*** *when* $\mathcal{M}[\![\mathsf{t}_1; ...; \mathsf{t}_i; \mathsf{t}_{i+1}; ...; \mathsf{t}_n]\!] \equiv_{\mathbf{D}} \mathcal{M}[\![\mathsf{t}_1; ...; \mathsf{t}_{i+1}; \mathsf{t}_i; ...; \mathsf{t}_n]\!]$.

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

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

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

5.7 Related Work

Our approach to semantics is similar to abstract interpretation: we have a concrete (exact) semantics and a family of abstractions parameterized by `z` (approximating semantics). We have not framed our approach this way because our approximations are not conservative, and would be difficult to formulate as abstractions when parameterized on a random source (which we intend to do).

Bayesian practitioners occasionally create languages for modeling and queries. Analyzing their properties is usually difficult, as they tend to be defined by implementations. Almost all of them compute converging approximations and support conditional queries. When they work as expected, they are useful.

Koller and Pfeffer [29] efficiently compute exact distributions for the outputs of programs in a Scheme-like language. BUGS [34] focuses on efficient approximate computation for probabilistic theories with a finitely many statements, with distributions that practitioners typically use. BLOG [37] exists specifically to allow stating distributions over countably infinite vectors. BLAISE [10] allows stating both distribution and approximation method for each random variable. Church [18] is a Scheme-like probabilistic language with approximate inference, and focuses on expressiveness.

Kiselyov [27] 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.

There is a fair amount of semantics work in probabilistic languages. Most of it is not motivated by Bayesian concerns, and thus does not define conditioning. Kozen [30] defines the meaning of bounded-space, imperative "while" programs as functions from probability measures to probability measures. Hurd [23] proves properties about programs with binary random choice by encoding programs and portions of measure theory in HOL.

Jones [24] develops a domain-theoretic variation of probability theory, and with it defines the probability monad, whose discrete version is a distribution-valued variation of the set or list monad. Ramsey and Pfeffer [46] 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 probability space, making it difficult to reason about conditioning.

Pfeffer also develops IBAL [45], apparently the only lambda calculus with finite probabilistic choice that also defines conditional queries. Park [42] extends a lambda calculus with probabilistic choice, defining it for a very general class of probability measures using inverse transform sampling.

5.8 Conclusions and Future Work

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.

Now that we are satisfied that our approach works, we turn our attention to uncountable sample spaces and theories with infinitely many statements.

Following measure-theoretic structure in our preliminary work should make the transition to uncountable spaces fairly smooth. The functional structure of the exact semantics will not change, but some details will. The random variable idiom will be identical, but will require measurability proofs. We will still interpret statements as state monad computations, but with general probability spaces as state instead of discrete probability spaces. We will use regular conditional probability in $\mathsf{cond}_{\mathsf{ps}}$, $\mathsf{extend}_{\mathsf{ps}}$ will calculate product σ-algebras and transition kernel products, and $\mathsf{dist}_{\mathsf{ps}}$ will return probability measures. We will not need to change $\mathcal{R}[\![\cdot]\!]$, $\mathbf{D}[\![\cdot]\!]$ or $\mathbf{P}[\![\cdot]\!]$. Many approximations are available; the most efficient and general

are sampling methods. We will likely choose sampling methods that parallelize easily.

The most general constructive way to specify theories with infinitely many primitive random variables is with recursive abstractions, but it is not clear what kind of abstraction we need. Lambdas are suitable for most functional programming, in which it is usually good that intermediate values are unobservable. However, they do not meet Bayesian needs: practitioners define theories to study them, not to obtain single answers. If lambdas were the only abstraction, returning every intermediate value from every lambda would become *good practice*. Because we do not know what form abstraction will take, we will likely develop it independently by allowing theories with infinitely many statements.

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 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.

Chapter 6

Interlude: Infinite Programs

Chapter 7

Preimage Computation: Running Probabilistic 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

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

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

7.1.1 Probability Densities

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

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

$$\begin{aligned} \mathsf{t}' \;:=\; & \mathsf{let}\;\; \mathsf{t} := \mathsf{normal}\ \mu\ 1 \\ & \mathsf{in}\;\; \mathsf{max}\ 0\ (\mathsf{min}\ 100\ \mathsf{t}) \end{aligned} \tag{7.1}$$

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

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

$$p_x(x \mid y) \;=\; \frac{p_y(y \mid x) \cdot \pi_x(x)}{\int p_y(y \mid x) \cdot \pi_x(x)\ dx} \tag{7.2}$$

Bayesians interpret probabilistic processes as defining p_y and π_x, and use (7.2) to find the distribution of "x given $y = c$." Even though "x given $x + y = 0$" has perfectly sensible distribution, Bayes' law for densities cannot express it.

7.1.2 Probability Measures

Measure-theoretic probability [28] is widely believed to be able to define every reasonable distribution that densities cannot. It mainly does this by *assigning probabilities to sets* instead of *assigning changes in probability to values*. Functions that do so are probability **measures**. In contrast to densities, probabilities of sets of values with different dimension *are* comparable, and probability measures *can* be defined on infinite products.

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

$$\Pr[B] \;=\; P(f^{-1}(B)) \tag{7.3}$$

The preimage $f^{-1}(B) = \{a \in X \mid f(a) \in B\}$ is the subset of X for which f yields a value in B, and is well-defined for any f. In the thermometer example (7.1), f would be an

interpretation of the program as a function, X would be the set of all random sources, and Y would be $\mathbb{R}$.

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

$$\Pr[B' \mid B] \;=\; \Pr[B' \cap B] \;/\; \Pr[B] \tag{7.4}$$

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

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

7.1.3 Measure-Theoretic Semantics

Most purely functional languages allow only nontermination as a side effect, and not probabilistic choice. Programmers therefore encode probabilistic programs as functions from random sources to outputs. Monads and other categorical classes such as idioms (i.e. applicative functors) can make doing so easier [23, 48].

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

1. Preimages can be defined only for functions with observable domains, which excludes lambdas.
2. If subsets of X and Y must be measurable, taking preimages under f must preserve

measurability (we say f itself is measurable). Proving the conditions under which this is true is difficult, especially if f may not terminate.

3. It is very difficult to define probability measures for arbitrary spaces of measurable functions [5].

Implementing a language based on such a semantics is complicated because

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

We address 1 and 4 by developing our semantics in λ_{ZFC} [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 it in Section 7.8. The outcome is worth it: we prove all probabilistic programs are measurable, regardless of the inputs on which they do not terminate. This includes uncomputable programs; for example, those that contain real equality tests and limits. We believe this result is the first of its kind, and is general enough to apply to almost all past and future work on probabilistic programming languages. To maintain the flow of this chapter, however, we put it off until Chapter 9.

For difficulty 3, we have discovered that the "first-orderness" of arrows [22] is a perfect fit for the "first-orderness" of measure theory.

7.1.4 Arrow Solution Overview

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

- A semantic function which, like the arrow calculus semantic function [33], transforms first-order programs into the computations of an arbitrary arrow.
- Arrows for evaluating expressions in different ways.

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

$$\begin{array}{ccccc} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{map}}} & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \\ {\scriptstyle \eta_{\perp *}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{map}*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}*}} \\ \mathsf{X} \rightsquigarrow_{\perp *} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{map}*}]{} & \mathsf{X} \underset{\mathsf{map}*}{\rightsquigarrow} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}*}]{} & \mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow} \mathsf{Y} \end{array} \tag{7.5}$$

From top-left to top-right, $\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ arrow computations are intensional functions that may raise errors, $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ instances produce extensional functions, and $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ instances compute preimages. Instances of arrows in the bottom row are like those in the top, except they thread an infinite store of random values, and can be constructed to always terminate. Most of our correctness theorems rely on proofs that every morphism in (7.5) is a homomorphism.

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

7.2 Arrows and First-Order Semantics

Like monads [55] and idioms [36], arrows [22] thread effects through computations in a way that imposes structure. But arrow computations are always

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

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

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 $\mathsf{first_a}$, we define $(\mathsf{\&\&\&_a})$. This combinator is typically called **fanout**, but its use will be clearer if we call it **pairing**. One way to strengthen an arrow a is to define an additional combinator $\mathsf{left_a}$, which can be used to choose an arrow computation based on the result of another. Again, we define a different combinator, $\mathsf{ifte_a}$ ("if-then-else").

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

$$\begin{aligned}
\mathsf{arr\ f} &:= \mathsf{f} \\
(\mathsf{f_1} \ggg \mathsf{f_2})\ \mathsf{a} &:= \mathsf{f_2\ (f_1\ a)} \\
(\mathsf{f_1\ \&\&\&\ f_2})\ \mathsf{a} &:= \langle \mathsf{f_1\ a}, \mathsf{f_2}, \mathsf{a} \rangle \\
\mathsf{ifte\ f_1\ f_2\ f_3\ a} &:= \mathsf{if\ (f_1\ a)\ (f_2\ a)\ (f_3\ a)} \\
\mathsf{lazy\ f\ a} &:= \mathsf{f\ 0\ a}
\end{aligned} \tag{7.6}$$

and try to define the following recursive function:

$$\mathsf{halt\text{-}on\text{-}true} := \mathsf{ifte\ (arr\ id)\ (arr\ id)\ halt\text{-}on\text{-}true} \tag{7.7}$$

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

$$\mathsf{halt\text{-}on\text{-}true} := \mathsf{ifte\ (arr\ id)\ (arr\ id)\ (lazy\ \lambda 0.\ halt\text{-}on\text{-}true)} \tag{7.8}$$

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

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

$$\begin{aligned}
\mathsf{arr}_{\mathsf{a}} &: (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\
(\ggg_{\mathsf{a}}) &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_{\mathsf{a}} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{z}) \\
(\&\&\&_{\mathsf{a}}) &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \langle \mathsf{y}, \mathsf{z} \rangle) \\
\mathsf{ifte}_{\mathsf{a}} &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\
\mathsf{lazy}_{\mathsf{a}} &: (1 \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})
\end{aligned} \tag{7.9}$$

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

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

Definition 7.2 (arrow homomorphism)**.** *A function* $\mathsf{lift}_{\mathsf{b}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *is an* ***arrow homomorphism*** *from arrow* a *to arrow* b *if the following distributive laws hold for appropriately typed* f*,* f_1*,* f_2 *and* f_3*:*

$$\mathsf{lift}_{\mathsf{b}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}) \equiv \mathsf{arr}_{\mathsf{b}}\ \mathsf{f} \tag{7.10}$$

$$\mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{f}_2) \equiv (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_1) \ggg_{\mathsf{b}} (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_2) \tag{7.11}$$

$$\mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}_1 \&\&\&_{\mathsf{a}} \mathsf{f}_2) \equiv (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_1) \&\&\&_{\mathsf{b}} (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_2) \tag{7.12}$$

$$\mathsf{lift}_{\mathsf{b}}\ (\mathsf{ifte}_{\mathsf{a}}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3) \equiv \mathsf{ifte}_{\mathsf{b}}\ (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_1)\ (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_2)\ (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_3) \tag{7.13}$$

$$\mathsf{lift}_{\mathsf{b}}\ (\mathsf{lazy}_{\mathsf{a}}\ \mathsf{f}) \equiv \mathsf{lazy}_{\mathsf{b}}\ \lambda 0.\, \mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}\ 0) \tag{7.14}$$

The arrow homomorphism laws state that $\mathsf{arr}_{\mathsf{a}} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$ must be a homomorphism from the function arrow (7.6) to arrow a. Roughly, arrow computations that do not use additional combinators can be transformed into $\mathsf{arr}_{\mathsf{a}}$ applied to a pure computation. They must be *function-like*.

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

over effectful:

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

$$(\mathsf{arr}_\mathsf{a}\ \mathsf{f}_1 \mathbin{\&\&\&_\mathsf{a}} \mathsf{f}_2) \ggg_\mathsf{a} \mathsf{arr}_\mathsf{a}\ \mathsf{snd} \equiv \mathsf{f}_2 \tag{7.16}$$

$$\mathsf{arr}_\mathsf{a}\ \mathsf{f}_1 \ggg_\mathsf{a} (\mathsf{f}_2 \mathbin{\&\&\&_\mathsf{a}} \mathsf{f}_3) \equiv (\mathsf{arr}_\mathsf{a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2) \mathbin{\&\&\&_\mathsf{a}} (\mathsf{arr}_\mathsf{a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_3) \tag{7.17}$$

$$\begin{aligned} \mathsf{arr}_\mathsf{a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{ifte}_\mathsf{a}\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{f}_4 \equiv\ & \mathsf{ifte}_\mathsf{a}\ (\mathsf{arr}_\mathsf{a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2) \\ & (\mathsf{arr}_\mathsf{a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_3) \\ & (\mathsf{arr}_\mathsf{a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_4) \end{aligned} \tag{7.18}$$

$$\mathsf{arr}_\mathsf{a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{lazy}_\mathsf{a}\ \mathsf{f}_2 \equiv \mathsf{lazy}_\mathsf{a}\ \lambda 0.\, \mathsf{arr}_\mathsf{a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2\ 0 \tag{7.19}$$

Equivalence between different arrow representations is usually proved in a strongly normalizing λ-calculus [32, 33], 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 (7.19) from existing arrow laws. We follow Hughes's reasoning [22] 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 (7.16), which *can* be derived from existing arrow laws, is a more problematic, in nonstrict λ-calculii as well as λ_{ZFC}. If f_1 can loop, using (7.16) to transform a computation can turn a nonterminating expression into a terminating one, or vice-versa. We could condition the pair extraction law on f_1's termination. 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* $\mathsf{lift}_\mathsf{b} : (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{b} \mathsf{y})$ *that has a right inverse is an* ***arrow epimorphism*** *from* a *to* b*.*

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

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

$$
\begin{array}{rcl}
[\![x := e;\ \ldots\ ;\ e_b]\!]_{\mathsf{a}} & :\equiv & x := [\![e]\!]_{\mathsf{a}}\,;\ \ldots\ ;\ [\![e_b]\!]_{\mathsf{a}} \\
[\![x\ e]\!]_{\mathsf{a}} & :\equiv & [\![\langle e, \langle\rangle\rangle]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} x \\
[\![\langle e_1, e_2\rangle]\!]_{\mathsf{a}} & :\equiv & [\![e_1]\!]_{\mathsf{a}} \mathbin{\&\&\&_{\mathsf{a}}} [\![e_2]\!]_{\mathsf{a}} \\
[\![\mathsf{fst}\ e]\!]_{\mathsf{a}} & :\equiv & [\![e]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{fst} \\
[\![\mathsf{snd}\ e]\!]_{\mathsf{a}} & :\equiv & [\![e]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \\
[\![v]\!]_{\mathsf{a}} & :\equiv & \mathsf{arr}_{\mathsf{a}}\ (\mathsf{const}\ v) \\
\mathsf{id} & := & \lambda \mathsf{a}.\, \mathsf{a} \\
\mathsf{const}\ \mathsf{b} & := & \lambda \mathsf{a}.\, \mathsf{b}
\end{array}
\qquad
\begin{array}{rcl}
[\![\mathsf{let}\ e\ e_b]\!]_{\mathsf{a}} & :\equiv & ([\![e]\!]_{\mathsf{a}} \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{id}) \ggg_{\mathsf{a}} [\![e_b]\!]_{\mathsf{a}} \\
[\![\mathsf{env}\ 0]\!]_{\mathsf{a}} & :\equiv & \mathsf{arr}_{\mathsf{a}}\ \mathsf{fst} \\
[\![\mathsf{env}\ (n+1)]\!]_{\mathsf{a}} & :\equiv & \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} [\![\mathsf{env}\ n]\!]_{\mathsf{a}} \\
[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]_{\mathsf{a}} & :\equiv & \mathsf{ifte}_{\mathsf{a}}\ [\![e_c]\!]_{\mathsf{a}}\ [\![\mathsf{lazy}\ e_t]\!]_{\mathsf{a}}\ [\![\mathsf{lazy}\ e_f]\!]_{\mathsf{a}} \\
[\![\mathsf{lazy}\ e]\!]_{\mathsf{a}} & :\equiv & \mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\, [\![e]\!]_{\mathsf{a}} \\
\multicolumn{3}{c}{\text{subject to } [\![p]\!]_{\mathsf{a}} : \langle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y} \text{ for some } \mathsf{y}}
\end{array}
$$

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

Proof. For the pair extraction law (7.16), rewrite in terms of $\mathsf{lift}_{\mathsf{b}}$, apply homomorphism laws, and apply the pair extraction law for arrow a:

$$
\begin{aligned}
&(\mathsf{arr}_{\mathsf{b}}\ \mathsf{f}_1 \mathbin{\&\&\&_{\mathsf{b}}} \mathsf{f}_2) \ggg_{\mathsf{b}} \mathsf{arr}_{\mathsf{b}}\ \mathsf{snd} && (7.20)\\
&\quad\equiv\ (\mathsf{lift}_{\mathsf{b}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1) \mathbin{\&\&\&_{\mathsf{b}}} (\mathsf{lift}_{\mathsf{b}}\ (\mathsf{lift}_{\mathsf{b}}^{-1}\ \mathsf{f}_2))) \ggg_{\mathsf{b}} \mathsf{arr}_{\mathsf{b}}\ \mathsf{snd} \\
&\quad\equiv\ \mathsf{lift}_{\mathsf{b}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1 \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{lift}_{\mathsf{b}}^{-1}\ \mathsf{f}_2) \ggg_{\mathsf{b}} \mathsf{lift}_{\mathsf{b}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}) \\
&\quad\equiv\ \mathsf{lift}_{\mathsf{b}}\ ((\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1 \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{lift}_{\mathsf{b}}^{-1}\ \mathsf{f}_2) \ggg_{\mathsf{b}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}) \\
&\quad\equiv\ \mathsf{lift}_{\mathsf{b}}\ (\mathsf{lift}_{\mathsf{b}}^{-1}\ \mathsf{f}_2) \\
&\quad\equiv\ \mathsf{f}_2
\end{aligned}
$$

The proofs for every other law are similar. □

7.2.2 First-Order Let-Calculus Semantics

Figure 7.1 defines a transformation from a first-order let-calculus to arrow computations for any arrow a. A program is a sequence of definition statements followed by a final expression. The semantic function $[\![\cdot]\!]_{\mathsf{a}}$ transforms each defining expression and the final expression into

arrow computations. Functions are named, but local variables and arguments are not. Instead, variables are referred to by De Bruijn indexes, with 0 referring to the innermost binding.

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

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

Definition 7.5 (well-defined expression)**.** *An expression e is **well-defined** under arrow a if $[\![e]\!]_{\mathsf{a}} : \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ for some x and y, and $[\![e]\!]_{\mathsf{a}}$ terminates.*

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

Theorem 7.6 (homomorphisms distribute over expressions)**.** *Let $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ be an arrow homomorphism. For all e, $[\![e]\!]_{\mathsf{b}} \equiv \mathsf{lift_b}\ [\![e]\!]_{\mathsf{a}}$.*

Proof. By structural induction. Bases cases proceed by expansion and using $\mathsf{arr_b} \equiv \mathsf{lift_b} \circ \mathsf{arr_a}$ (7.10). For example, for constants:

$$\begin{aligned} [\![v]\!]_{\mathsf{b}} &\equiv \mathsf{arr_b}\ (\mathsf{const}\ v) \\ &\equiv \mathsf{lift_b}\ (\mathsf{arr_a}\ (\mathsf{const}\ v)) \\ &\equiv \mathsf{lift_b}\ [\![v]\!]_{\mathsf{a}} \end{aligned} \tag{7.21}$$

Inductive cases proceed by expansion, applying the inductive hypothesis on subterms, and

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

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

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

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

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

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

Figure 7.2: Bottom arrow definitions.

applying distributive laws (7.11)–(7.14). For example, for pairing:

$$\begin{aligned} [\![\langle e_1, e_2 \rangle]\!]_\mathsf{b} &\equiv [\![e_1]\!]_\mathsf{b} \ \&\&\&_\mathsf{b} \ [\![e_2]\!]_\mathsf{b} \\ &\equiv (\mathsf{lift}_\mathsf{b} \ [\![e_1]\!]_\mathsf{a}) \ \&\&\&_\mathsf{b} \ (\mathsf{lift}_\mathsf{b} \ [\![e_2]\!]_\mathsf{a}) \\ &\equiv \mathsf{lift}_\mathsf{b} \ ([\![e_1]\!]_\mathsf{a} \ \&\&\&_\mathsf{a} \ [\![e_2]\!]_\mathsf{a}) \\ &\equiv \mathsf{lift}_\mathsf{b} \ [\![\langle e_1, e_2 \rangle]\!]_\mathsf{a} \end{aligned} \tag{7.22}$$

It is not hard to check the remaining cases. □

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, $[\![\cdot]\!]_\mathsf{b}$ is correct.

7.3 The Bottom Arrow

Using the diagram in (7.5) as a sort of map, we start in the upper-left corner:

$$\begin{array}{ccccc} \mathsf{X} \rightsquigarrow_\perp \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{map}}} & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \\ \downarrow{\scriptstyle \eta_{\perp^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{map}^*}]{} & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}^*}]{} & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{array} \tag{7.23}$$

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

To use Theorem 7.6 to prove correct the interpretations of expressions as preimage

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

$$\mathsf{arr}_{\mathsf{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Y})$$
$$\mathsf{arr}_{\mathsf{map}} \ := \ \mathsf{lift}_{\mathsf{map}} \circ \mathsf{arr}_{\perp}$$

$$(\ggg_{\mathsf{map}}) : (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Y}) \Rightarrow (\mathsf{Y} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Z}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Z})$$
$$(\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A} \ := \ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}$$
$$\mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1')$$
$$\mathsf{in}\ \ \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1'$$

$$(\&\&\&_{\mathsf{map}}) : (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \langle \mathsf{Y}_1, \mathsf{Y}_2 \rangle)$$
$$(\mathsf{g}_1 \ \&\&\&_{\mathsf{map}}\ \mathsf{g}_2)\ \mathsf{A} \ := \ \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A} \rangle_{\mathsf{map}}$$

$$\mathsf{ifte}_{\mathsf{map}} : (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Bool}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Y})$$
$$\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A} \ := \ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}$$
$$\mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\})$$
$$\mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\})$$
$$\mathsf{in}\ \ \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3'$$

$$\mathsf{lazy}_{\mathsf{map}} : (1 \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Y})) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Y})$$
$$\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}\ \mathsf{A} \ := \ \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A})$$

$$\mathsf{lift}_{\mathsf{map}} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptstyle \mathsf{map}} \mathsf{Y})$$
$$\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A} := \{\langle \mathsf{a}, \mathsf{b} \rangle \in \mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \mid \mathsf{b} \neq \perp\}$$

Figure 7.3: Mapping arrow definitions.

arrow computations, we need the preimage arrow to be homomorphic to a simpler arrow with easily understood behavior. The function arrow (7.6) is an obvious candidate. However, we will need to explicitly handle nontermination as an error value, so we need a slightly more complicated arrow.

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

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

Definition 7.7 (bottom arrow equivalence)**.** *Two computations* $\mathsf{f}_1 : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ *and* $\mathsf{f}_2 : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ *are equivalent, or* $\mathsf{f}_1 \equiv \mathsf{f}_2$*, when* $\mathsf{f}_1\ \mathsf{a} \equiv \mathsf{f}_2\ \mathsf{a}$ *for all* $\mathsf{a} \in \mathsf{X}$*.*

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

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

7.4 Deriving the Mapping Arrow

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

produce mappings or are mappings themselves.

It is tempting to try to make the mapping arrow's computations mapping-valued; i.e. $\mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y} ::= \mathsf{X} \rightharpoonup \mathsf{Y}$, with $\mathsf{f}_1 \ggg_{\mathsf{map}} \mathsf{f}_2 := \mathsf{f}_2 \circ_{\mathsf{map}} \mathsf{f}_1$ and $\mathsf{f}_1 \mathbin{\&\&\&}_{\mathsf{map}} \mathsf{f}_2 := \langle \mathsf{f}_1, \mathsf{f}_2 \rangle_{\mathsf{map}}$. Unfortunately, we could not define $\mathsf{arr}_{\mathsf{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$: to define a mapping, we need a domain, but lambdas' domains are unobservable.

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

$$\mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y} \ ::= \ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{7.24}$$

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

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

$$\begin{aligned} &\mathsf{domain}_{\bot} : (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ \mathsf{X} \\ &\mathsf{domain}_{\bot}\ \mathsf{f}\ \mathsf{A} \ := \ \{\mathsf{a} \in \mathsf{A} \mid \mathsf{f}\ \mathsf{a} \neq \bot\} \end{aligned} \tag{7.25}$$

$$\begin{aligned} &\mathsf{lift}_{\mathsf{map}} : (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}) \\ &\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A} \ := \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_{\bot}\ \mathsf{f}\ \mathsf{A}) \end{aligned} \tag{7.26}$$

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

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

Definition 7.9 (mapping arrow equivalence)**.** *Two computations* $\mathsf{g}_1 : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$

are equivalent, or $\mathsf{g}_1 \equiv \mathsf{g}_2$*, when* $\mathsf{g}_1\ \mathsf{A} \equiv \mathsf{g}_2\ \mathsf{A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$.

Clearly $\mathsf{arr}_{\mathsf{map}} := \mathsf{lift}_{\mathsf{map}} \circ \mathsf{arr}_{\perp}$ meets the first homomorphism law (7.10). The remainder of this section derives $(\&\&\&_{\mathsf{map}})$, $(\ggg_{\mathsf{map}})$, $\mathsf{ifte}_{\mathsf{map}}$ and $\mathsf{lazy}_{\mathsf{map}}$ from bottom arrow combinators, in a way that ensures $\mathsf{lift}_{\mathsf{map}}$ is an arrow homomorphism. Figure 7.3 contains the resulting definitions.

7.4.1 Composition

Starting with the left side of (7.11), we expand definitions, simplify f by restricting it to a set for which $\mathsf{f}_1\ \mathsf{a} \neq \perp$, and substitute f's definition:

$$\begin{aligned}
\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1 \ggg \mathsf{f}_2)\ \mathsf{A} \equiv\ & \mathsf{let}\quad \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp)\ \perp\ (\mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a})) \\
& \qquad \mathsf{A}' := \mathsf{domain}_{\perp}\ \mathsf{f}\ \mathsf{A} \\
& \quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}' \\
\equiv\ & \mathsf{let}\quad \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a}) \\
& \qquad \mathsf{A}' := \mathsf{domain}_{\perp}\ \mathsf{f}\ (\mathsf{domain}_{\perp}\ \mathsf{f}_1\ \mathsf{A}) \\
& \quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}' \\
\equiv\ & \mathsf{let}\ \ \mathsf{A}' := \{\mathsf{a} \in \mathsf{domain}_{\perp}\ \mathsf{f}_1\ \mathsf{A} \mid \mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a}) \neq \perp\} \\
& \quad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\, \mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a})
\end{aligned} \tag{7.27}$$

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

$$\begin{aligned}
\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1 \ggg \mathsf{f}_2)\ \mathsf{A} \equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
& \qquad \mathsf{A}' := \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{domain}_{\perp}\ \mathsf{f}_2\ (\mathsf{range}\ \mathsf{g}_1)) \\
& \quad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\, \mathsf{f}_2\ (\mathsf{g}_1\ \mathsf{a}) \\
\equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
& \qquad \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{range}\ \mathsf{g}_1) \\
& \qquad \mathsf{A}' := \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{domain}\ \mathsf{g}_2) \\
& \quad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\, \mathsf{g}_2\ (\mathsf{g}_1\ \mathsf{a}) \\
\equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
& \qquad \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{range}\ \mathsf{g}_1) \\
& \quad \mathsf{in}\ \ \mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1
\end{aligned} \tag{7.28}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$ and g_2 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$ gives a definition for $(\ggg_{\mathsf{map}})$ (Figure 7.3) for which (7.11) holds.

7.4.2 Pairing

Starting with the left side of (7.12), we expand definitions and replace the definition of A' with one that does not depend on f:

$$\begin{aligned}
\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1 \mathbin{\&\&\&_\perp} \mathsf{f}_2)\ \mathsf{A} \ \equiv\ & \mathsf{let} \quad \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{a} = \perp)\ \perp\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a} \rangle \\
& \qquad\ \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A} \\
& \quad\ \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}' \\
\equiv\ & \mathsf{let}\ \ \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f}_1\ \mathsf{A} \cap \mathsf{domain}_\perp\ \mathsf{f}_2\ \mathsf{A} \\
& \quad\ \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\, \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a} \rangle
\end{aligned} \tag{7.29}$$

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

$$\begin{aligned}
\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1 \mathbin{\&\&\&_\perp} \mathsf{f}_2)\ \mathsf{A} \ \equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
& \qquad\ \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A} \\
& \qquad\ \mathsf{A}' := \mathsf{domain}\ \mathsf{g}_1 \cap \mathsf{domain}\ \mathsf{g}_2 \\
& \quad\ \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\, \langle \mathsf{g}_1\ \mathsf{a}, \mathsf{g}_2\ \mathsf{a} \rangle \\
\equiv\ & \langle \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A}, \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A} \rangle_{\mathsf{map}}
\end{aligned} \tag{7.30}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$ and g_2 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$ gives a definition for $(\&\&\&_{\mathsf{map}})$ (Figure 7.3) for which (7.12) holds.

7.4.3 Conditional

Starting with the left side of (7.13), we expand definitions, and simplify f by restricting it to a domain for which $\mathsf{f}_1\ \mathsf{a} \neq \bot$:

$$\begin{aligned}
\mathsf{lift}_{\mathsf{map}}\ (\mathsf{ifte}_\bot\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{A} \equiv\ & \mathsf{let}\ \ \mathsf{f} := \lambda\,\mathsf{a}.\,\mathsf{case}\ \ \mathsf{f}_1\ \mathsf{a} \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathsf{true}\ \ \longrightarrow\ \mathsf{f}_2\ \mathsf{a} \\
& \qquad\qquad\qquad\qquad\qquad\quad \mathsf{false}\ \ \longrightarrow\ \mathsf{f}_3\ \mathsf{a} \\
& \qquad\qquad\qquad\qquad\qquad\quad \bot\ \ \longrightarrow\ \bot \\
& \qquad\qquad\quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\bot\ \mathsf{f}\ \mathsf{A}) \\
\equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \\
& \qquad\quad\ \mathsf{A}_2 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\} \\
& \qquad\quad\ \mathsf{A}_3 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\} \\
& \qquad\qquad \mathsf{f} := \lambda\,\mathsf{a}.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a}) \\
& \qquad\quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\bot\ \mathsf{f}\ (\mathsf{A}_2 \uplus \mathsf{A}_3))
\end{aligned} \tag{7.31}$$

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

$$\begin{aligned}
\mathsf{lift}_{\mathsf{map}}\ (\mathsf{ifte}_\bot\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{A} \equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
& \qquad\quad\ \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}) \\
& \qquad\quad\ \mathsf{g}_3 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}) \\
& \qquad\quad\ \mathsf{A}' := \mathsf{domain}\ \mathsf{g}_2 \uplus \mathsf{domain}\ \mathsf{g}_3 \\
& \qquad\ \mathsf{in}\ \ \lambda\,\mathsf{a} \in \mathsf{A}'.\,\mathsf{if}\ (\mathsf{a} \in \mathsf{domain}\ \mathsf{g}_2)\ (\mathsf{g}_2\ \mathsf{a})\ (\mathsf{g}_3\ \mathsf{a}) \\
\equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
& \qquad\quad\ \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}) \\
& \qquad\quad\ \mathsf{g}_3 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}) \\
& \qquad\ \mathsf{in}\ \ \mathsf{g}_2 \uplus_{\mathsf{map}} \mathsf{g}_3
\end{aligned} \tag{7.32}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$, g_2 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$, and g_3 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3$ gives a definition for $\mathsf{ifte}_{\mathsf{map}}$ (Figure 7.3) for which (7.13) holds.

7.4.4 Laziness

Starting with the left side of (7.14), we expand definitions:

$$\begin{aligned}\mathsf{lift}_{\mathsf{map}}\ (\mathsf{lazy}_\perp\ \mathsf{f})\ \mathsf{A}\ \equiv\ &\mathsf{let}\ \ \mathsf{A}' := \mathsf{domain}_\perp\ (\lambda\mathsf{a}.\,\mathsf{f}\ 0\ \mathsf{a})\ \mathsf{A}\\ &\mathsf{in}\ \ \mathsf{mapping}\ (\lambda\mathsf{a}.\,\mathsf{f}\ 0\ \mathsf{a})\ \mathsf{A}'\end{aligned} \tag{7.33}$$

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

$$\begin{aligned}\mathsf{lift}_{\mathsf{map}}\ (\mathsf{lazy}_\perp\ \mathsf{f})\ \mathsf{A}\ &\equiv\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{mapping}\ (\mathsf{f}\ 0)\ (\mathsf{domain}_\perp\ (\mathsf{f}\ 0)\ \mathsf{A}))\\ &\equiv\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}\ 0)\ \mathsf{A})\end{aligned} \tag{7.34}$$

Substituting $\mathsf{g}\ 0$ for $\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}\ 0)$ gives a $\mathsf{lazy}_{\mathsf{map}}$ (Figure 7.3) for which (7.14) holds.

7.4.5 Correctness

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

Proof. By construction. □

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

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

$$\begin{aligned}&\mathsf{nonmonotone} : \mathsf{Bool} \mathrel{\underset{\mathsf{map}}{\rightsquigarrow}} \mathsf{Bool}\\ &\mathsf{nonmonotone}\ \mathsf{A}\ :=\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{mapping}\ \mathsf{id}\ \mathsf{Bool})\ \varnothing\end{aligned} \tag{7.35}$$

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

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

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

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

$$\mathsf{domain}_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y}) \Rightarrow \mathsf{Set\ X}$$
$$\mathsf{domain}_{\mathsf{pre}}\ \langle \mathsf{Y}', \mathsf{p} \rangle \ := \ \mathsf{p}\ \mathsf{Y}'$$

$$\mathsf{range}_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y}) \Rightarrow \mathsf{Set\ Y}$$
$$\mathsf{range}_{\mathsf{pre}}\ \langle \mathsf{Y}', \mathsf{p} \rangle \ := \ \mathsf{Y}'$$

$$\langle \cdot, \cdot \rangle_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y}_1 \times \mathsf{Y}_2)$$
$$\langle \langle \mathsf{Y}'_1, \mathsf{p}_1 \rangle, \langle \mathsf{Y}'_2, \mathsf{p}_2 \rangle \rangle_{\mathsf{pre}} \ := \ \mathsf{let}\ \ \mathsf{Y}' := \mathsf{Y}'_1 \times \mathsf{Y}'_2$$
$$\mathsf{p} := \lambda \mathsf{B}. \bigcup_{\langle \mathsf{b}_1, \mathsf{b}_2 \rangle \in \mathsf{B}} (\mathsf{p}_1\ \{\mathsf{b}_1\}) \cap (\mathsf{p}_2\ \{\mathsf{b}_2\})$$
$$\mathsf{in}\ \ \langle \mathsf{Y}', \mathsf{p} \rangle$$

$$(\circ_{\mathsf{pre}}) : (\mathsf{Y} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Z})$$
$$\langle \mathsf{Z}', \mathsf{p}_2 \rangle \circ_{\mathsf{pre}} \mathsf{h}_1 \ := \ \langle \mathsf{Z}', \lambda \mathsf{C}.\, \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ (\mathsf{p}_2\ \mathsf{C}) \rangle$$

$$(\uplus_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y})$$
$$\mathsf{h}_1 \uplus_{\mathsf{pre}} \mathsf{h}_2 \ := \ \mathsf{let}\ \ \mathsf{Y}' := (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1) \cup (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_2)$$
$$\mathsf{p} := \lambda \mathsf{B}. (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{B}) \uplus (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_2\ \mathsf{B})$$
$$\mathsf{in}\ \ \langle \mathsf{Y}', \mathsf{p} \rangle$$

Figure 7.4: Lazy preimage mappings and operations.

proving the arrow laws simple.

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

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

Theorem 7.13. $\mathsf{lift}_{\mathsf{map}}$ *is an arrow epimorphism.*

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

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

7.5 Lazy Preimage Mappings

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

If we wish to compute with infinite sets in the language implementation, we will need an abstraction that makes it easy to replace computation on points with computation on

sets whose representations allow efficient operations. Therefore, in the preimage arrow, we confine computation on points to instances of

$$\mathsf{X} \mathrel{\underset{\mathsf{pre}}{\rightharpoonup}} \mathsf{Y} \;::=\; \langle \mathsf{Set}\ \mathsf{Y}, \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X} \rangle \tag{7.36}$$

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

Converting a mapping to a lazy preimage mapping requires constructing a delayed application of preimage:

$$\begin{aligned} &\mathsf{pre} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \mathrel{\underset{\mathsf{pre}}{\rightharpoonup}} \mathsf{Y}) \\ &\mathsf{pre}\ \mathsf{g} \;:=\; \langle \mathsf{range}\ \mathsf{g}, \lambda \mathsf{B}.\, \mathsf{preimage}\ \mathsf{g}\ \mathsf{B} \rangle \end{aligned} \tag{7.37}$$

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

$$\begin{aligned} &\mathsf{ap}_{\mathsf{pre}} : (\mathsf{X} \mathrel{\underset{\mathsf{pre}}{\rightharpoonup}} \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X} \\ &\mathsf{ap}_{\mathsf{pre}}\ \langle \mathsf{Y}', \mathsf{p} \rangle\ \mathsf{B} \;:=\; \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y}') \end{aligned} \tag{7.38}$$

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

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

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

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

Figure 7.4 defines more operations on preimage mappings, including pairing, composition, and disjoint union operations corresponding to the mapping operations in Figure 4.1. To prove them correct, we need preimage mappings to be equivalent when they compute the same preimages.

Definition 7.17 (preimage mapping equivalence)**.** $\mathsf{h}_1 : \mathsf{X} \rightleftharpoons_{\mathsf{pre}} \mathsf{Y}$ *and* $\mathsf{h}_2 : \mathsf{X} \rightleftharpoons_{\mathsf{pre}} \mathsf{Y}$ *are equivalent, or* $\mathsf{h}_1 \equiv \mathsf{h}_2$*, when* $\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{B} \equiv \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_2\ \mathsf{B}$ *for all* $\mathsf{B} \subseteq \mathsf{Y}$*.*

Similarly to proving arrows correct, we prove the operations in Figure 7.4 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 $(\circ_{\mathsf{map}})$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{Y} \rightharpoonup \mathsf{Z}$*. For all* $\mathsf{C} \subseteq \mathsf{Z}$*,* $\mathsf{preimage}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{C} = \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{C})$*.*

Theorem 7.19 (pre distributes over $(\circ_{\mathsf{map}})$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{Y} \rightharpoonup \mathsf{Z}$*. Then* $\mathsf{pre}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1) \equiv (\mathsf{pre}\ \mathsf{g}_2) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_1)$*.*

Proof. Let $\langle \mathsf{Z}', \mathsf{p}_2 \rangle := \mathsf{pre}\ \mathsf{g}_2$ and $\mathsf{C} \subseteq \mathsf{Z}$. Starting from the right side, expand definitions, apply Theorem 7.16, apply Lemma 7.18, and apply Theorem 7.16 again:

$$
\begin{aligned}
& \mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre}\ \mathsf{g}_2) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_1))\ \mathsf{C} && (7.39)\\
& \equiv\ \mathsf{let}\ \ \mathsf{h} := \lambda\, \mathsf{C}.\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{p}_2\ \mathsf{C})\\
&\qquad\ \ \mathsf{in}\ \ \mathsf{h}\ (\mathsf{C} \cap \mathsf{Z}')\\
& \equiv\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{p}_2\ (\mathsf{C} \cap \mathsf{Z}'))\\
& \equiv\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_2)\ \mathsf{C})\\
& \equiv\ \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{C})\\
& \equiv\ \mathsf{preimage}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{C}\\
& \equiv\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1))\ \mathsf{C}
\end{aligned}
$$

□

7.5.2 Pairing

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

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

Theorem 7.21 (pre distributes over $\langle\cdot,\cdot\rangle_{\mathsf{map}}$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. Then* $\mathsf{pre}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}} \equiv \langle \mathsf{pre}\ \mathsf{g}_1, \mathsf{pre}\ \mathsf{g}_2\rangle_{\mathsf{pre}}$.

Proof. Let $\langle \mathsf{Y}_1', \mathsf{p}_1\rangle := \mathsf{pre}\ \mathsf{g}_1$, $\langle \mathsf{Y}_2', \mathsf{p}_2\rangle := \mathsf{pre}\ \mathsf{g}_2$ and $\mathsf{B} \in \mathsf{Y}_1 \times \mathsf{Y}_2$. Starting from the right side, expand definitions, apply Theorem 7.16, apply Lemma 7.20, note that a product of singletons is a singleton pair, distribute preimage over the union, apply Lemma 7.15, and apply Theorem 7.16 again:

$$
\begin{aligned}
&\mathsf{ap}_{\mathsf{pre}}\ \langle \mathsf{pre}\ \mathsf{g}_1, \mathsf{pre}\ \mathsf{g}_2\rangle_{\mathsf{pre}}\ \mathsf{B} && (7.40)\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{Y}' := \mathsf{Y}_1' \times \mathsf{Y}_2'\\
&\qquad\qquad\ \mathsf{p} := \lambda \mathsf{B}.\ \bigcup_{\langle \mathsf{y}_1, \mathsf{y}_2\rangle \in \mathsf{B}} (\mathsf{p}_1\ \{\mathsf{y}_1\}) \cap (\mathsf{p}_2\ \{\mathsf{y}_2\})\\
&\qquad\quad\ \mathsf{in}\ \ \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y}')\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1, \mathsf{y}_2\rangle \in \mathsf{B} \cap (\mathsf{Y}_1' \times \mathsf{Y}_2')} (\mathsf{p}_1\ \{\mathsf{y}_1\}) \cap (\mathsf{p}_2\ \{\mathsf{y}_2\})\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1, \mathsf{y}_2\rangle \in \mathsf{B} \cap (\mathsf{Y}_1' \times \mathsf{Y}_2')} (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{y}_1\}) \cap (\mathsf{preimage}\ \mathsf{g}_2\ \{\mathsf{y}_2\})\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1, \mathsf{y}_2\rangle \in \mathsf{B} \cap (\mathsf{Y}_1' \times \mathsf{Y}_2')} (\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ (\{\mathsf{y}_1\} \times \{\mathsf{y}_2\}))\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1, \mathsf{y}_2\rangle \in \mathsf{B} \cap (\mathsf{Y}_1' \times \mathsf{Y}_2')} (\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ \{\langle \mathsf{y}_1, \mathsf{y}_2\rangle\})\\
&\quad\equiv\ \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ (\mathsf{B} \cap (\mathsf{Y}_1' \times \mathsf{Y}_2'))\\
&\quad\equiv\ \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ \mathsf{B}\\
&\quad\equiv\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}})\ \mathsf{B}
\end{aligned}
$$

□

7.5.3 Disjoint Union

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

Lemma 7.22 (preimage distributes over $(\uplus_{\mathsf{map}})$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *have disjoint domains. For all* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{B} = (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}) \uplus (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B})$.

Theorem 7.23 (pre distributes over $(\uplus_{\mathsf{map}})$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *have disjoint domains. Then* $\mathsf{pre}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2) \equiv (\mathsf{pre}\ \mathsf{g}_1) \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_2)$.

Proof. Let $\mathsf{Y}'_1 := \mathsf{range}\ \mathsf{g}_1$, $\mathsf{Y}'_2 := \mathsf{range}\ \mathsf{g}_2$ and $\mathsf{B} \subseteq \mathsf{Y}$. Starting from the right side, expand definitions, apply Theorem 7.16, apply Lemma 7.22, appy Lemma 7.15, and apply Theorem 7.16 again:

$$
\begin{aligned}
&\mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre}\ \mathsf{g}_1) \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_2))\ \mathsf{B} && (7.41)\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{Y}' := \mathsf{Y}'_1 \cup \mathsf{Y}'_2\\
&\qquad\qquad\ \mathsf{h} := \lambda\, \mathsf{B}.\, (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ \mathsf{B}) \uplus (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_2)\ \mathsf{B})\\
&\qquad\quad\ \mathsf{in}\ \ \mathsf{h}\ (\mathsf{B} \cap \mathsf{Y}')\\
&\quad\equiv\ (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2))) \uplus (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_2)\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)))\\
&\quad\equiv\ (\mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2))) \uplus (\mathsf{preimage}\ \mathsf{g}_2\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)))\\
&\quad\equiv\ \mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2))\\
&\quad\equiv\ \mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{B}\\
&\quad\equiv\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2))\ \mathsf{B}
\end{aligned}
$$

□

7.6 Deriving the Preimage Arrow

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

As with the mapping arrow and mappings, we cannot have $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{X} \underset{\mathsf{pre}}{\rightarrow} \mathsf{Y}$: we run into trouble trying to define $\mathsf{arr}_{\mathsf{pre}}$ because a preimage mapping needs an observable range.

To get one, it is easiest to parameterize preimage computations on a $\mathsf{Set\ X}$; therefore the ***preimage arrow*** type constructor is

$$\mathsf{X} \overset{\rightsquigarrow}{_{\mathsf{pre}}} \mathsf{Y} \ ::= \ \mathsf{Set\ X} \Rightarrow (\mathsf{X} \overset{\rightharpoonup}{_{\mathsf{pre}}} \mathsf{Y}) \tag{7.42}$$

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

To use Theorem 7.6, we need to define correctness using a lift from the mapping arrow to the preimage arrow. A simple candidate with the right type is

$$\begin{aligned} &\mathsf{lift_{pre}} : (\mathsf{X} \overset{\rightsquigarrow}{_{\mathsf{map}}} \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{_{\mathsf{pre}}} \mathsf{Y}) \\ &\mathsf{lift_{pre}\ g\ A} \ := \ \mathsf{pre\ (g\ A)} \end{aligned} \tag{7.43}$$

By $\mathsf{lift_{pre}}$'s definition and Theorem 7.16, for all $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{_{\mathsf{map}}} \mathsf{Y}$, $\mathsf{A} \subseteq \mathsf{X}$ and $\mathsf{B} \subseteq \mathsf{Y}$,

$$\begin{aligned} \mathsf{ap_{pre}\ (lift_{pre}\ g\ A)\ B} \ &\equiv \ \mathsf{ap_{pre}\ (pre\ (g\ A))\ B} \\ &\equiv \ \mathsf{preimage\ (g\ A)\ B} \end{aligned} \tag{7.44}$$

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

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

Definition 7.24 (preimage arrow equivalence)**.** *Two computations* $\mathsf{h_1} : \mathsf{X} \overset{\rightsquigarrow}{_{\mathsf{pre}}} \mathsf{Y}$ *and* $\mathsf{h_2} : \mathsf{X} \overset{\rightsquigarrow}{_{\mathsf{pre}}} \mathsf{Y}$ *are equivalent, or* $\mathsf{h_1} \equiv \mathsf{h_2}$*, when* $\mathsf{h_1\ A} \equiv \mathsf{h_2\ A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$*.*

As with $\mathsf{arr_{map}}$, defining $\mathsf{arr_{pre}}$ as a composition meets (7.10). The remainder of this section derives $(\mathsf{\&\&\&_{pre}})$, $(\ggg_{\mathsf{pre}})$, $\mathsf{ifte_{pre}}$ and $\mathsf{lazy_{pre}}$ from mapping arrow combinators, in a way that ensures $\mathsf{lift_{pre}}$ is an arrow homomorphism from the mapping arrow to the preimage arrow. Figure 7.5 contains the resulting definitions.

$$X \overset{\rightsquigarrow}{\scriptstyle pre} Y ::= \mathsf{Set}\ X \Rightarrow (X \overset{\rightarrow}{\scriptstyle pre} Y)$$

$$\mathsf{arr}_{pre} : (X \Rightarrow Y) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle pre} Y)$$
$$\mathsf{arr}_{pre}\ := \ \mathsf{lift}_{pre} \circ \mathsf{arr}_{map}$$

$$(\ggg_{pre}) : (X \overset{\rightsquigarrow}{\scriptstyle pre} Y) \Rightarrow (Y \overset{\rightsquigarrow}{\scriptstyle pre} Z) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle pre} Z)$$
$$(h_1 \ggg_{pre} h_2)\ A\ :=\ \mathsf{let}\ \ h_1' := h_1\ A \\ \qquad\qquad\qquad\qquad\qquad\qquad h_2' := h_2\ (\mathsf{range}_{pre}\ h_1') \\ \qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ h_2' \circ_{pre} h_1'$$

$$(\&\&\&_{pre}) : (X \overset{\rightsquigarrow}{\scriptstyle pre} Y) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle pre} Z) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle pre} Y \times Z)$$
$$(h_1 \ \&\&\&_{pre}\ h_2)\ A\ :=\ \langle h_1\ A, h_2\ A\rangle_{pre}$$

$$\mathsf{ifte}_{pre} : (X \overset{\rightsquigarrow}{\scriptstyle pre} \mathsf{Bool}) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle pre} Y) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle pre} Y) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle pre} Y)$$
$$\mathsf{ifte}_{pre}\ h_1\ h_2\ h_3\ A\ :=\ \mathsf{let}\ \ h_1' := h_1\ A \\ \qquad\qquad\qquad\qquad\qquad\qquad h_2' := h_2\ (\mathsf{ap}_{pre}\ h_1'\ \{\mathsf{true}\}) \\ \qquad\qquad\qquad\qquad\qquad\qquad h_3' := h_3\ (\mathsf{ap}_{pre}\ h_1'\ \{\mathsf{false}\}) \\ \qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ h_2' \uplus_{pre} h_3'$$

$$\mathsf{lazy}_{pre} : (1 \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle pre} Y)) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle pre} Y)$$
$$\mathsf{lazy}_{pre}\ h\ A\ :=\ \mathsf{if}\ (A = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (h\ 0\ A)$$

$$\mathsf{lift}_{pre} : (X \overset{\rightsquigarrow}{\scriptstyle map} Y) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle pre} Y)$$
$$\mathsf{lift}_{pre}\ g\ A\ :=\ \mathsf{pre}\ (g\ A)$$

Figure 7.5: Preimage arrow definitions.

7.6.1 Pairing

Starting with the left side of (7.12), we expand definitions, apply Theorem 7.21, and rewrite in terms of lift_{pre}:

$$\begin{aligned} \mathsf{ap}_{pre}\ (\mathsf{lift}_{pre}\ (g_1 \ \&\&\&_{map}\ g_2)\ A)\ B &\equiv \mathsf{ap}_{pre}\ (\mathsf{pre}\ \langle g_1\ A, g_2\ A\rangle_{map})\ B \\ &\equiv \mathsf{ap}_{pre}\ \langle \mathsf{pre}\ (g_1\ A), \mathsf{pre}\ (g_2\ A)\rangle_{pre}\ B \\ &\equiv \mathsf{ap}_{pre}\ \langle \mathsf{lift}_{pre}\ g_1\ A, \mathsf{lift}_{pre}\ g_2\ A\rangle_{pre}\ B \end{aligned} \tag{7.45}$$

Substituting h_1 for $\mathsf{lift}_{pre}\ g_1$ and h_2 for $\mathsf{lift}_{pre}\ g_2$, and removing the application of ap_{pre} from both sides of the equivalence gives a definition of $(\&\&\&_{pre})$ (Figure 7.5) for which (7.12) holds.

7.6.2 Composition

Starting with the left side of (7.11), we expand definitions, apply Theorem 7.19 and rewrite in terms of lift_{pre}:

$$\begin{aligned} \mathsf{ap}_{pre}\ (\mathsf{lift}_{pre}\ (g_1 \ggg_{map} g_2)\ A)\ C \equiv\ &\mathsf{let}\ \ g_1' := g_1\ A \\ &\qquad g_2' := g_2\ (\mathsf{range}\ g_1') \\ &\quad \mathsf{in}\ \ \mathsf{ap}_{pre}\ (\mathsf{pre}\ (g_2' \circ_{map} g_1'))\ C \end{aligned} \tag{7.46}$$

$$
\begin{aligned}
&\equiv \ \mathsf{let}\ \ \mathsf{g}'_1 := \mathsf{g}_1\ \mathsf{A} \\
&\qquad\ \ \mathsf{g}'_2 := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}'_1) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre}\ \mathsf{g}'_1) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}'_2))\ \mathsf{C} \\
&\equiv \ \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} \\
&\qquad\ \ \mathsf{h}_2 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2\ (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1)\ \mathsf{C}
\end{aligned}
$$

Substituting h_1 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1$ and h_2 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2$, and removing the application of $\mathsf{ap}_{\mathsf{pre}}$ from both sides of the equivalence gives a definition of $(\ggg_{\mathsf{pre}})$ (Figure 7.5) for which (7.11) holds.

7.6.3 Conditional

Starting with the left side of (7.13), we expand terms, apply Theorem 7.23, rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$, and apply Theorem 7.16 in h_2 and h_3:

$$
\begin{aligned}
\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3)\ \mathsf{A})\ \mathsf{B} \ &\equiv \ \mathsf{let}\ \ \mathsf{g}'_1 := \mathsf{g}_1\ \mathsf{A} \\
&\qquad\ \ \mathsf{g}'_2 := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}'_1\ \{\mathsf{true}\}) \\
&\qquad\ \ \mathsf{g}'_3 := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}'_1\ \{\mathsf{false}\}) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}'_2 \uplus_{\mathsf{map}} \mathsf{g}'_3))\ \mathsf{B} \\
&\equiv \ \mathsf{let}\ \ \mathsf{g}'_1 := \mathsf{g}_1\ \mathsf{A} \\
&\qquad\ \ \mathsf{g}'_2 := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}'_1\ \{\mathsf{true}\}) \\
&\qquad\ \ \mathsf{g}'_3 := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}'_1\ \{\mathsf{false}\}) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre}\ \mathsf{g}'_2) \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}'_3))\ \mathsf{B} \\
&\equiv \ \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} \\
&\qquad\ \ \mathsf{h}_2 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2\ (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \{\mathsf{true}\}) \\
&\qquad\ \ \mathsf{h}_3 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_3\ (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \{\mathsf{false}\}) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}_2 \uplus_{\mathsf{pre}} \mathsf{h}_3)\ \mathsf{B}
\end{aligned}
\tag{7.47}
$$

Substituting h_1 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1$, h_2 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2$ and h_3 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_3$, and removing the application of $\mathsf{ap}_{\mathsf{pre}}$ from both sides of the equivalence gives a definition of $\mathsf{ifte}_{\mathsf{pre}}$ (Figure 7.5) for which (7.13) holds.

7.6.4 Laziness

Starting with the left side of (7.14), expand definitions, distribute pre over the branches of if, and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}\ 0)$:

$$\begin{aligned}
\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g})\ \mathsf{A})\ \mathsf{B} \equiv\ & \mathsf{let}\ \ \mathsf{g}' := \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}) \\
& \quad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}')\ \mathsf{B} \\
\equiv\ & \mathsf{let}\ \ \mathsf{h} := \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (\mathsf{pre}\ (\mathsf{g}\ 0\ \mathsf{A})) \\
& \quad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{B} \\
\equiv\ & \mathsf{let}\ \ \mathsf{h} := \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}\ 0)\ \mathsf{A}) \\
& \quad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{B}
\end{aligned} \tag{7.48}$$

Substituting $\mathsf{h}\ 0$ for $\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}\ 0)$ and removing the application of $\mathsf{ap}_{\mathsf{pre}}$ from both sides of the equivalence gives a definition for $\mathsf{lazy}_{\mathsf{pre}}$ (Figure 7.5) for which (7.14) holds.

7.6.5 Correctness

Theorem 7.25 (preimage arrow correctness)**.** $\mathsf{lift}_{\mathsf{pre}}$ *is a homomorphism.*

Proof. By construction. □

Corollary 7.26 (semantic correctness)**.** *For all* e, $[\![e]\!]_{\mathsf{pre}} \equiv \mathsf{lift}_{\mathsf{pre}}\ [\![e]\!]_{\mathsf{map}}$.

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

Definition 7.27 (preimage arrow law)**.** *Let* $\mathsf{h} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{\rightarrow}} \mathsf{Y}$*. If there exists a* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{\rightarrow}} \mathsf{Y}$ *such that* $\mathsf{h} \equiv \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}$*, then* h *obeys the* ***preimage arrow law****.*

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

Theorem 7.28. $\mathsf{lift}_{\mathsf{pre}}$ *is an arrow epimorphism.*

Proof. Follows from Theorem 7.25 and restriction of $\mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y}$ to instances for which the preimage arrow law (Definition 7.27) holds. □

Corollary 7.29. $\mathsf{arr}_{\mathsf{pre}}$, $(\&\&\&_{\mathsf{pre}})$, $(\ggg_{\mathsf{pre}})$, $\mathsf{ifte}_{\mathsf{pre}}$ *and* $\mathsf{lazy}_{\mathsf{pre}}$ *define an arrow.*

7.7 Preimages Under Partial, Probabilistic Functions

We have defined everything on the top of our roadmap:

$$\begin{array}{ccccc} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{map}}} & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \\ {\scriptstyle \eta_{\perp^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{map}^*}]{} & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}^*}]{} & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{array} \tag{7.49}$$

and proved that $\mathsf{lift}_{\mathsf{map}}$ and $\mathsf{lift}_{\mathsf{pre}}$ are homomorphisms. Now we move down from all three top arrows simultaneously, and prove every morphism in (7.49) is an arrow homomorphism.

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 $\mathsf{random} < \mathsf{p}$:

$$\mathsf{geometric\ p} \ := \ \mathsf{if\ (random} < \mathsf{p)\ 0\ (1 + geometric\ p)} \tag{7.50}$$

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

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

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

- Determine how random indexes numbers in r.

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

7.7.2 Threading and Indexing

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

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

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

Definition 7.30 (binary indexing scheme)**.** *Let* J *be an index set,* $\mathsf{j}_0 \in \mathsf{J}$ *a distinguished element, and* $\mathsf{left} : \mathsf{J} \Rightarrow \mathsf{J}$ *and* $\mathsf{right} : \mathsf{J} \Rightarrow \mathsf{J}$ *be total, injective functions. If for all* $\mathsf{j} \in \mathsf{J}$*,* $\mathsf{j} = \mathsf{next}\ \mathsf{j}_0$ *for some finite composition* next *of* left *and* right*, then* J*,* j_0*,* left *and* right *define a* ***binary indexing scheme****.*

For example, let J be the set of lists of $\{0, 1\}$, $\mathsf{j}_0 := \langle\rangle$, and $\mathsf{left}\ \mathsf{j} := \langle 0, \mathsf{j}\rangle$ and $\mathsf{right}\ \mathsf{j} := \langle 1, \mathsf{j}\rangle$. Alternatively, let J be the set of dyadic rationals in $(0, 1)$ (i.e. those with power-of-two denominators), $\mathsf{j}_0 := \frac{1}{2}$ and

$$\begin{aligned} \mathsf{left}\ (\mathsf{p}/\,\mathsf{q}) &:= (\mathsf{p} - \tfrac{1}{2})/\,\mathsf{q} \\ \mathsf{right}\ (\mathsf{p}/\,\mathsf{q}) &:= (\mathsf{p} + \tfrac{1}{2})/\,\mathsf{q} \end{aligned} \tag{7.51}$$

$$
\begin{array}{l}
\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} \ ::= \ \mathsf{AStore\ s\ (x} \rightsquigarrow_{\mathsf{a}} \mathsf{y)} \ ::= \ \mathsf{J} \Rightarrow (\langle \mathsf{s}, \mathsf{x} \rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\
\\
\mathsf{arr}_{\mathsf{a}^*} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\
\mathsf{arr}_{\mathsf{a}^*} \ := \ \eta_{\mathsf{a}^*} \circ \mathsf{arr}_{\mathsf{a}} \\
\\
(\ggg_{\mathsf{a}^*}) : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_{\mathsf{a}^*} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{z}) \\
(\mathsf{k}_1 \ggg_{\mathsf{a}^*} \mathsf{k}_2)\ \mathsf{j} \ := \\
\quad (\mathsf{arr}_{\mathsf{a}}\ \mathsf{fst} \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{k}_1\ (\mathsf{left\ j})) \ggg_{\mathsf{a}} \mathsf{k}_2\ (\mathsf{right\ j}) \\
\\
(\mathbin{\&\&\&_{\mathsf{a}^*}}) : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}_1) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}_2) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \langle \mathsf{y}_1, \mathsf{y}_2 \rangle) \\
(\mathsf{k}_1 \mathbin{\&\&\&_{\mathsf{a}^*}} \mathsf{k}_2)\ \mathsf{j} \ := \ \mathsf{k}_1\ (\mathsf{left\ j}) \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{k}_2\ (\mathsf{right\ j})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{ifte}_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\
\mathsf{ifte}_{\mathsf{a}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} \ := \ \mathsf{ifte}_{\mathsf{a}}\ (\mathsf{k}_1\ (\mathsf{left\ j})) \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{k}_2\ (\mathsf{left\ (right\ j)})) \\
\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{k}_3\ (\mathsf{right\ (right\ j)})) \\
\\
\mathsf{lazy}_{\mathsf{a}^*} : (1 \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\
\mathsf{lazy}_{\mathsf{a}^*}\ \mathsf{k}\ \mathsf{j} \ := \ \mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\, \mathsf{k}\ 0\ \mathsf{j} \\
\hline
\eta_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\
\eta_{\mathsf{a}^*}\ \mathsf{f}\ \mathsf{j} \ := \ \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}
\end{array}
$$

Figure 7.6: AStore (associative store) arrow transformer definitions.

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

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

7.7.3 Applicative, Associative Store Transformer

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

$$\mathsf{AStore\ s\ (x} \rightsquigarrow_{\mathsf{a}} \mathsf{y)} \ ::= \ \mathsf{J} \Rightarrow (\langle \mathsf{s}, \mathsf{x} \rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \tag{7.52}$$

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

on to the original computation, ignoring j:

$$\begin{aligned} &\eta_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow \mathsf{AStore\ s\ } (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\ &\eta_{\mathsf{a}^*}\ \mathsf{f\ j} \ := \ \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f} \end{aligned} \tag{7.53}$$

Figure 7.6 defines the remaining combinators. Each subcomputation receives left j, right j, or some other unique binary index. We thus think of programs interpreted as AStore arrows as being completely unrolled into an infinite binary tree, with each expression labeled with its tree index.

7.7.4 Partial, Probabilistic Programs

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

Definition 7.31 (random source)**.** *Let* $\mathsf{R} := \mathsf{J} \to [0,1]$*. A* ***random source*** *is any infinite binary tree* $\mathsf{r} \in \mathsf{R}$*.*

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

Definition 7.32 (branch trace)**.** *A* ***branch trace*** *is any* $\mathsf{t} \in \mathsf{J} \to \mathsf{Bool}_\perp$ *such that* $\mathsf{t\ j} = \mathsf{true}$ *or* $\mathsf{t\ j} = \mathsf{false}$ *for no more than finitely many* $\mathsf{j} \in \mathsf{J}$*.*

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

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

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

defined:

$$\begin{aligned}
&\mathsf{random}_{\mathsf{a}^*} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} [0,1] \\
&\mathsf{random}_{\mathsf{a}^*}\ \mathsf{j} \ :=\ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{fst} \ggg \pi\ \mathsf{j}) \\
&[\![\mathsf{random}]\!]_{\mathsf{a}^*} \ :\equiv\ \mathsf{random}_{\mathsf{a}^*}
\end{aligned} \tag{7.54}$$

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

$$\begin{aligned}
&\mathsf{branch}_{\mathsf{a}^*} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool} \\
&\mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j} \ :=\ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{snd} \ggg \pi\ \mathsf{j}) \\
&\\
&\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\
&\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} \ :=\ \mathsf{ifte}_{\mathsf{a}}\ ((\mathsf{k}_1\ (\mathsf{left}\ \mathsf{j}) \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j}) \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{agrees}) \\
&\qquad\qquad\qquad\qquad (\mathsf{k}_2\ (\mathsf{left}\ (\mathsf{right}\ \mathsf{j}))) \\
&\qquad\qquad\qquad\qquad (\mathsf{k}_3\ (\mathsf{right}\ (\mathsf{right}\ \mathsf{j})))
\end{aligned} \tag{7.55}$$

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

$$[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*} \ :\equiv\ \mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ [\![e_c]\!]^{\Downarrow}_{\mathsf{a}^*}\ [\![\mathsf{lazy}\ e_t]\!]^{\Downarrow}_{\mathsf{a}^*}\ [\![\mathsf{lazy}\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*} \tag{7.56}$$

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

Definition 7.33 (terminating, probabilistic arrows)**.** *Define*

$$\begin{aligned}
\mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y} \ &::=\ \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \\
\mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} \ &::=\ \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\
\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \ &::=\ \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})
\end{aligned} \tag{7.57}$$

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

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

Theorem 7.35 (pure AStore arrow correctness)**.** η_{a^*} *is a homomorphism.*

Proof. Defining $\mathsf{arr}_{\mathsf{a}^*}$ as a composition clearly meets the first homomorphism law (7.10). For homomorphism laws (7.11)–(7.13), start from the right side, expand definitions, and use arrow laws (7.16)–(7.18) to factor out $\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}$.

For (7.14), additionally β-expand within the outer thunk, then use the lazy distributive law (7.19) to extract $\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}$. □

Corollary 7.36 (pure semantic correctness)**.** *For all pure* e*,* $[\![e]\!]_{\mathsf{a}^*} \equiv \eta_{\mathsf{a}^*}\ [\![e]\!]_{\mathsf{a}}$*.*

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

$$\begin{aligned} &\mathsf{lift}_{\mathsf{b}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y}) \\ &\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{f}\ \mathsf{j}\ :=\ \mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}\ \mathsf{j}) \end{aligned} \tag{7.58}$$

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

$$\begin{array}{ccc} \mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y} & \xrightarrow{\mathsf{lift_b}} & \mathsf{x} \rightsquigarrow_\mathsf{b} \mathsf{y} \\ {\scriptstyle \eta_{\mathsf{a}^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{b}^*}} \\ \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} & \xrightarrow[\mathsf{lift_{b^*}}]{} & \mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y} \end{array} \tag{7.59}$$

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

Theorem 7.37 (natural transformation)**.** *If* $\mathsf{lift_b}$ *is an arrow homomorphism, then* (7.59) *commutes.*

Proof. Expand definitions and apply homomorphism laws (7.11) and (7.10) for $\mathsf{lift_b}$:

$$\begin{aligned} \mathsf{lift_{b^*}}\ (\eta_{\mathsf{a}^*}\ \mathsf{f}) &\equiv \lambda \mathsf{j}.\, \mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{snd} \ggg_\mathsf{a} \mathsf{f}) \\ &\equiv \lambda \mathsf{j}.\, \mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{snd}) \ggg_\mathsf{b} \mathsf{lift_b}\ \mathsf{f} \\ &\equiv \lambda \mathsf{j}.\, \mathsf{arr_b}\ \mathsf{snd} \ggg_\mathsf{b} \mathsf{lift_b}\ \mathsf{f} \\ &\equiv \eta_{\mathsf{b}^*}\ (\mathsf{lift_b}\ \mathsf{f}) \end{aligned} \tag{7.60}$$

□

Theorem 7.38 (effectful AStore arrow correctness)**.** *If* $\mathsf{lift_b}$ *is an arrow homomorphism from* a *to* b*, then* $\mathsf{lift_{b^*}}$ *is an arrow homomorphism from* a^* *to* b^**.*

Proof. For each homomorphism property (7.10)–(7.14), expand the definitions of $\mathsf{lift_{b^*}}$ and the combinator, distribute $\mathsf{lift_b}$, rewrite in terms of $\mathsf{lift_{b^*}}$, and rewrite using the definition of the combinator. For example, for distribution over pairing:

$$\begin{aligned} \mathsf{lift_{b^*}}\ (\mathsf{k}_1 \mathbin{\&\&\&_{\mathsf{a}^*}} \mathsf{k}_2)\ \mathsf{j} &\equiv \mathsf{lift_b}\ ((\mathsf{k}_1 \mathbin{\&\&\&_{\mathsf{a}^*}} \mathsf{k}_2)\ \mathsf{j}) \\ &\equiv \mathsf{lift_b}\ (\mathsf{k}_1\ (\mathsf{left}\ \mathsf{j}) \mathbin{\&\&\&_\mathsf{a}} \mathsf{k}_2\ (\mathsf{right}\ \mathsf{j})) \\ &\equiv \mathsf{lift_b}\ (\mathsf{k}_1\ (\mathsf{left}\ \mathsf{j})) \mathbin{\&\&\&_\mathsf{b}} \mathsf{lift_b}\ (\mathsf{k}_2\ (\mathsf{right}\ \mathsf{j})) \\ &\equiv (\mathsf{lift_{b^*}}\ \mathsf{k}_1)\ (\mathsf{left}\ \mathsf{j}) \mathbin{\&\&\&_\mathsf{b}} (\mathsf{lift_{b^*}}\ \mathsf{k}_2)\ (\mathsf{right}\ \mathsf{j}) \\ &\equiv (\mathsf{lift_{b^*}}\ \mathsf{k}_1 \mathbin{\&\&\&_{\mathsf{b}^*}} \mathsf{lift_{b^*}}\ \mathsf{k}_2)\ \mathsf{j} \end{aligned} \tag{7.61}$$

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

Corollary 7.39 (effectful semantic correctness)**.** *If* $\mathsf{lift}_{\mathsf{b}}$ *is an arrow homomorphism, then for all expressions* e, $[\![e]\!]_{\mathsf{b}^*} \equiv \mathsf{lift}_{\mathsf{b}^*}\ [\![e]\!]_{\mathsf{a}^*}$ *and* $[\![e]\!]^{\Downarrow}_{\mathsf{b}^*} \equiv \mathsf{lift}_{\mathsf{b}^*}\ [\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$.

Corollary 7.40 (mapping* and preimage* arrow correctness)**.** *The following diagram commutes:*

$$\begin{array}{ccccc} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{map}}} & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \\ {\scriptstyle \eta_{\perp^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{map}^*}]{} & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}^*}]{} & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{array} \tag{7.62}$$

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

Corollary 7.41 (effectful semantic correctness)**.** *For all expressions* e*,*

$$\begin{aligned} [\![e]\!]_{\mathsf{pre}^*} &\equiv \mathsf{lift}_{\mathsf{pre}^*}\ (\mathsf{lift}_{\mathsf{map}^*}\ [\![e]\!]_{\perp^*}) \\ [\![e]\!]^{\Downarrow}_{\mathsf{pre}^*} &\equiv \mathsf{lift}_{\mathsf{pre}^*}\ (\mathsf{lift}_{\mathsf{map}^*}\ [\![e]\!]^{\Downarrow}_{\perp^*}) \end{aligned} \tag{7.63}$$

7.7.6 Termination

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

Definition 7.42 (maximal domain)**.** *A computation's* ***maximal domain*** *is the largest* A^* *for which*

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

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

Because the above statements imply termination, A^* is a subset of the largest domain for which the computations terminate. It is not too hard to show (but is a bit tedious) that

lifting computations preserves the maximal domain; e.g. the maximal domain of $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}$ is the same as f's, and the maximal domain of $\mathsf{lift}_{\mathsf{pre}^*}\ \mathsf{g}$ is the same as g's.

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

Theorem 7.43 (mapping arrow restriction)**.** *Let* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$*, and* $\mathsf{A}^{\Downarrow} \subseteq \mathsf{X}$ *be the largest for which* $\mathsf{g}\ \mathsf{A}^{\Downarrow}$ *terminates. For all* $\mathsf{A} \subseteq \mathsf{A}^{\Downarrow}$*,* $\mathsf{g}\ \mathsf{A} = \mathsf{restrict}\ (\mathsf{g}\ \mathsf{A}^{\Downarrow})\ \mathsf{A}$*.*

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

$$\begin{aligned}
\mathsf{restrict}\ (\mathsf{g}\ \mathsf{A}^{\Downarrow})\ \mathsf{A} &\equiv \mathsf{restrict}\ (\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}^{\Downarrow})\ \mathsf{A} \\
&\equiv \mathsf{restrict}\ (\{\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}^{\Downarrow} \mid \mathsf{b} \neq \perp\})\ \mathsf{A} \\
&\equiv \{\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \mid \mathsf{b} \neq \perp\} \\
&\equiv \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A} \\
&\equiv \mathsf{g}\ \mathsf{A}
\end{aligned} \tag{7.64}$$

□

Theorem 7.44 (domain closure operators)**.** *If* $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$*,* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$ *and* $\mathsf{h} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}$*, then* $\mathsf{domain}_{\perp}\ \mathsf{f}$*,* $\mathsf{domain} \circ \mathsf{g}$*, and* $\mathsf{domain}_{\mathsf{pre}} \circ \mathsf{h}$ *are monotone, decreasing, and idempotent in the subdomains on which they terminate.*

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

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

Theorem 7.45. *Let* $\mathsf{f} := [\![e]\!]_{\perp^*} : \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}$ *with maximal domain* A^**, and* $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\perp^*}$*. For all* $\langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle \in \mathsf{A}^*$*, there exists a* $\mathsf{T}' \subseteq \mathsf{T}$ *such that*

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

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

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

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

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

Definition 7.46 (index prefix/suffix)**.** *A finite* $\mathsf{J}' \subset \mathsf{J}$ *is an* ***index prefix*** *if* $\mathsf{J}' = \{\mathsf{j}_0\}$ *or, for some index prefix* J'' *and* $\mathsf{j} \in \mathsf{J}''$, $\mathsf{J}' = \mathsf{J}'' \uplus \{\mathsf{left}\ \mathsf{j}\}$ *or* $\mathsf{J}' = \mathsf{J}'' \uplus \{\mathsf{right}\ \mathsf{j}\}$. *The corresponding* ***index suffix*** *is* $\mathsf{J}\backslash\mathsf{J}'$.

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

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

Theorem 7.47. *Let* $\mathsf{f} := [\![e]\!]_{\bot^*} : \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}$ *with maximal domain* A^*, *and* $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\bot^*}$. *For all* $\mathsf{a} \in ((\mathsf{R} \times \mathsf{T}) \times \mathsf{X})\backslash\mathsf{A}^*$, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \bot$.

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

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

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

Corollary 7.48. *For all e, the maximal domain of $[\![e]\!]^{\Downarrow}_{\bot^*}$ is a subset of that of $[\![e]\!]_{\bot^*}$.*

Corollary 7.49. *Let $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\bot^*} : \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}$ with maximal domain A^*, and $\mathsf{f} := [\![e]\!]_{\bot^*}$. For all $\mathsf{a} \in \mathsf{A}^*$, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \mathsf{f}\ \mathsf{j}_0\ \mathsf{a}$.*

Corollary 7.50 (correct computation everywhere)**.** *Let $[\![e]\!]^{\Downarrow}_{\bot^*} : \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}$ have maximal domain A^*, and $\mathsf{X}' := (\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$. For all $\mathsf{a} \in \mathsf{X}'$, $\mathsf{A} \subseteq \mathsf{X}'$ and $\mathsf{B} \subseteq \mathsf{Y}$,*

$$\begin{aligned}
[\![e]\!]^{\Downarrow}_{\bot^*}\ \mathsf{j}_0\ \mathsf{a} &= \mathsf{if}\ (\mathsf{a} \in \mathsf{A}^*)\ ([\![e]\!]_{\bot^*}\ \mathsf{j}_0\ \mathsf{a})\ \bot \\
[\![e]\!]^{\Downarrow}_{\mathsf{map}^*}\ \mathsf{j}_0\ \mathsf{A} &= [\![e]\!]_{\mathsf{map}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*) \\
\mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} &= \mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]_{\mathsf{pre}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*))\ \mathsf{B}
\end{aligned} \tag{7.65}$$

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

7.8 Output Probabilities and Measurability

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

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

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

Our main results are the best we could hope for. First, the interpretations of all expressions preserve measurability, regardless of nontermination.

$$
\begin{aligned}
\mathsf{id}_{\mathsf{pre}}\ \mathsf{A} &:= \langle \mathsf{A}, \lambda \mathsf{B}.\, \mathsf{B} \rangle \\
\mathsf{const}_{\mathsf{pre}}\ \mathsf{b}\ \mathsf{A} &:= \langle \{\mathsf{b}\}, \lambda \mathsf{B}.\, \mathsf{if}\ (\mathsf{B} = \varnothing)\ \varnothing\ \mathsf{A} \rangle \\
\mathsf{fst}_{\mathsf{pre}}\ \mathsf{A} &:= \langle \mathsf{proj}_1\ \mathsf{A}, \mathsf{unproj}_1\ \mathsf{A} \rangle \\
\mathsf{snd}_{\mathsf{pre}}\ \mathsf{A} &:= \langle \mathsf{proj}_2\ \mathsf{A}, \mathsf{unproj}_2\ \mathsf{A} \rangle \\
\pi_{\mathsf{pre}}\ \mathsf{j}\ \mathsf{A} &:= \langle \mathsf{proj}\ \mathsf{j}\ \mathsf{A}, \mathsf{unproj}\ \mathsf{j}\ \mathsf{A} \rangle
\end{aligned}
$$

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

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

$$
\begin{aligned}
&\mathsf{proj}_1 : \mathsf{Set}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \Rightarrow \mathsf{Set}\ \mathsf{X}_1 \\
&\mathsf{proj}_1 := \mathsf{image}\ \mathsf{fst}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{proj}_2 : \mathsf{Set}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \Rightarrow \mathsf{Set}\ \mathsf{X}_2 \\
&\mathsf{proj}_2 := \mathsf{image}\ \mathsf{snd}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{unproj}_1 : \mathsf{Set}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \Rightarrow \mathsf{Set}\ \mathsf{X}_1 \Rightarrow \mathsf{Set}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \\
&\mathsf{unproj}_1\ \mathsf{A}\ \mathsf{A}_1 := \mathsf{preimage}\ (\mathsf{mapping}\ \mathsf{fst}\ \mathsf{A})\ \mathsf{A}_1 \\
&\qquad\qquad \equiv \mathsf{A} \cap (\mathsf{A}_1 \times \mathsf{proj}_2\ \mathsf{A})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{unproj}_2 : \mathsf{Set}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \Rightarrow \mathsf{Set}\ \mathsf{X}_2 \Rightarrow \mathsf{Set}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \\
&\mathsf{unproj}_2\ \mathsf{A}\ \mathsf{A}_2 := \mathsf{preimage}\ (\mathsf{mapping}\ \mathsf{snd}\ \mathsf{A})\ \mathsf{A}_2 \\
&\qquad\qquad \equiv \mathsf{A} \cap (\mathsf{proj}_1\ \mathsf{A} \times \mathsf{A}_2)
\end{aligned}
$$

Figure 7.7: Preimage arrow lifts needed to interpret probabilistic programs.

Theorem 7.51. *For all expressions e,* $[\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ *preserves measurability.*

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

Theorem 7.52. *If* $\mathsf{A} \subseteq (\mathsf{R} \times \mathsf{T}) \times \{\langle\rangle\}$ *is measurable, then* $\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ \mathsf{A}$ *is measurable.*

The proofs of these theorems are in Chapter 9.

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

Fortunately, we need only certain lifts. Figure 7.1 (which defines $[\![\cdot]\!]_{\mathsf{a}}$) lifts id, $\mathsf{const\ b}$, fst and snd. Section 7.7.4, which defines the combinators used to interpret partial, probabilistic programs, lifts $\pi\ \mathsf{j}$ and agrees. Measurable functions made available as language primitives of course must be lifted to the preimage arrow.

Figure 7.7 gives explicit definitions for $\mathsf{arr_{pre}\ id}$, $\mathsf{arr_{pre}\ fst}$, $\mathsf{arr_{pre}\ snd}$, $\mathsf{arr_{pre}\ (const\ b)}$ and $\mathsf{arr_{pre}}\ (\pi\ \mathsf{j})$. (We will deal with agrees separately.) To implement them, we must model sets in a way that makes $\mathsf{A} = \varnothing$ is decidable, and the following representable and finitely computable:

- $\mathsf{A} \cap \mathsf{B}$, $\varnothing$, $\{\mathsf{true}\}$, $\{\mathsf{false}\}$ and $\{\mathsf{b}\}$ for every $\mathsf{const\ b}$
- $\mathsf{A}_1 \times \mathsf{A}_2$, $\mathsf{proj}_1\ \mathsf{A}$ and $\mathsf{proj}_2\ \mathsf{A}$ (7.66)
- $\mathsf{J} \to \mathsf{X}$, $\mathsf{proj\ j\ A}$ and $\mathsf{unproj\ j\ A\ B}$

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

Definition 7.53 (rectangular family)**.** $\mathsf{Rect\ X}$ *denotes the* ***rectangular family*** *of subsets of* X. $\mathsf{Rect\ X}$ *must contain* $\varnothing$ *and* X*, and be closed under finite intersections. Products must satisfy the following rules:*

$$\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \ = \ (\mathsf{Rect\ X}_1) \boxtimes (\mathsf{Rect\ X}_2) \tag{7.67}$$

$$\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X}) \ = \ (\mathsf{Rect\ X})^{\boxtimes \mathsf{J}} \tag{7.68}$$

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

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 \ := \ \{\mathsf{A}_1 \times \mathsf{A}_2 \mid \mathsf{A}_1 \in \mathcal{A}_1, \mathsf{A}_2 \in \mathcal{A}_2\} \tag{7.69}$$

$$\mathcal{A}^{\boxtimes \mathsf{J}} \ := \bigcup_{\mathsf{J}' \subset \mathsf{J}\ \mathit{finite}} \left\{ \prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_{\mathsf{j}} \;\middle|\; \mathsf{A}_{\mathsf{j}} \in \mathcal{A}, \mathsf{j} \in \mathsf{J}' \iff \mathsf{A}_{\mathsf{j}} \subset \bigcup \mathcal{A} \right\} \tag{7.70}$$

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

Further, all of the operations in (7.66) can be exactly implemented if finite sets are modeled directly, sets in ordered spaces (such as $\mathbb{R}$) are modeled by intervals, and sets in

$\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$ are modeled by pairs of type $\langle \mathsf{Rect}\ \mathsf{X}_1, \mathsf{Rect}\ \mathsf{X}_2 \rangle$. By (7.70), sets in $\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X})$ have no more than finitely many projections that are proper subsets of X. They can be modeled by *finite* binary trees, where unrepresented projections are implicitly X.

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

Theorem 7.54 (T model)**.** *If* $\mathsf{T}' \in \mathsf{Rect}\ (\mathsf{J} \to \mathsf{Bool}_\bot)$ *and* $\mathsf{j} \in \mathsf{J}$*, then* $\mathsf{proj}\ \mathsf{j}\ \mathsf{T}' = \mathsf{proj}\ \mathsf{j}\ (\mathsf{T}' \cap \mathsf{T})$*. If* $\mathsf{B} \subseteq \mathsf{Bool}_\bot$*, then* $\mathsf{unproj}\ \mathsf{j}\ (\mathsf{T}' \cap \mathsf{T})\ \mathsf{B} = \mathsf{unproj}\ \mathsf{j}\ \mathsf{T}'\ \mathsf{B} \cap \mathsf{T}$*.*

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

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

The statement about unproj is an easy corollary. □

7.9.2 Approximate Preimage Mapping Operations

Implementing $\mathsf{lazy}_{\mathsf{pre}}$ (defined in Figure 7.5) requires computing pre, but only for the empty mapping, which is trivial: $\mathsf{pre}\ \varnothing \equiv \langle \varnothing, \lambda\, \mathsf{B}.\, \varnothing \rangle$. Implementing the other combinators requires $(\circ_{\mathsf{pre}})$, $\langle \cdot, \cdot \rangle_{\mathsf{pre}}$ and $(\uplus_{\mathsf{pre}})$.

From the preimage mapping definitions (Figure 7.4), we see that $\mathsf{ap}_{\mathsf{pre}}$ is defined using $(\cap)$ and that $(\circ_{\mathsf{pre}})$ is defined using $\mathsf{ap}_{\mathsf{pre}}$, so $(\circ_{\mathsf{pre}})$ is directly implementable. Unfortunately, we hit a snag with $\langle \cdot, \cdot \rangle_{\mathsf{pre}}$: it loops over possibly uncountably many members of B in a big union. At this point, we need to approximate.

Theorem 7.55 (pair preimage approximation)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. For all* $\mathsf{B} \subseteq \mathsf{Y}_1 \times \mathsf{Y}_2$*,* $\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathsf{B} \subseteq \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{proj}_1\ \mathsf{B}) \cap \mathsf{preimage}\ \mathsf{g}_2\ (\mathsf{proj}_2\ \mathsf{B})$*.*

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

It is not hard to use Theorem 7.55 to show that

$$\begin{aligned}
&\langle\cdot,\cdot\rangle'_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_1 \times \mathsf{Y}_2) \\
&\langle\langle \mathsf{Y}'_1, \mathsf{p}_1\rangle, \langle \mathsf{Y}'_2, \mathsf{p}_2\rangle\rangle'_{\mathsf{pre}} \;:=\; \langle \mathsf{Y}'_1 \times \mathsf{Y}'_2, \lambda\,\mathsf{B}.\, \mathsf{p}_1\ (\mathsf{proj}_1\ \mathsf{B}) \cap \mathsf{p}_2\ (\mathsf{proj}_2\ \mathsf{B})\rangle
\end{aligned} \tag{7.71}$$

computes covering rectangles of preimages under pairing.

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

$$\begin{aligned}
(\mathsf{A}_1 \times \mathsf{A}_2) \vee (\mathsf{B}_1 \times \mathsf{B}_2) \;&=\; (\mathsf{A}_1 \vee \mathsf{B}_1) \times (\mathsf{A}_2 \vee \mathsf{B}_2) \\
(\textstyle\prod_{\mathsf{j}\in\mathsf{J}} \mathsf{A}_\mathsf{j}) \vee (\prod_{\mathsf{j}\in\mathsf{J}} \mathsf{B}_\mathsf{j}) \;&=\; \textstyle\prod_{\mathsf{j}\in\mathsf{J}} \mathsf{A}_\mathsf{j} \vee \mathsf{B}_\mathsf{j}
\end{aligned} \tag{7.72}$$

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

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

$$\begin{aligned}
&(\uplus'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\
&\mathsf{h}_1 \uplus'_{\mathsf{pre}} \mathsf{h}_2 \;:=\; \mathsf{let}\ \ \mathsf{Y}' := \mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1 \vee \mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_2 \\
&\qquad\qquad\qquad\quad \mathsf{p} := \lambda\,\mathsf{B}.\, \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{B} \vee \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_2\ \mathsf{B} \\
&\qquad\qquad\qquad\ \ \mathsf{in}\ \ \langle \mathsf{Y}', \mathsf{p}\rangle
\end{aligned} \tag{7.73}$$

To interpret programs that may not terminate, or that terminate with probability 1, we need to approximate $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$ (7.55), which is defined in terms of agrees. Defining its approximation in terms of an approximation of agrees would not allow us to preserve the fact that expressions interpreted using $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$ always terminate. The best approximation of the preimage of Bool under agrees (as a mapping) is $\mathsf{Bool} \times \mathsf{Bool}$, which contains $\langle\mathsf{true}, \mathsf{false}\rangle$ and $\langle\mathsf{false}, \mathsf{true}\rangle$, and thus would not constrain the test to agree with the branch trace.

A lengthy (elided) sequence of substitutions to the defining expression for $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$

results in an agrees-free equivalence:

$$\begin{aligned}
\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j}\ \mathsf{A}\ \equiv\ \mathsf{let}\ \ \langle \mathsf{C_k}, \mathsf{p_k}\rangle &:= \mathsf{k}_1\ \mathsf{j}_1\ \mathsf{A} \\
\langle \mathsf{C_b}, \mathsf{p_b}\rangle &:= \mathsf{branch}_{\mathsf{pre}^*}\ \mathsf{j}\ \mathsf{A} \\
\mathsf{C}_2 &:= \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{true}\} \\
\mathsf{C}_3 &:= \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{false}\} \\
\mathsf{A}_2 &:= \mathsf{p_k}\ \mathsf{C}_2 \cap \mathsf{p_b}\ \mathsf{C}_2 \\
\mathsf{A}_3 &:= \mathsf{p_k}\ \mathsf{C}_3 \cap \mathsf{p_b}\ \mathsf{C}_3 \\
\mathsf{in}\ \ \mathsf{k}_2\ \mathsf{j}_2\ \mathsf{A}_2 &\uplus_{\mathsf{pre}} \mathsf{k}_3\ \mathsf{j}_3\ \mathsf{A}_3
\end{aligned} \tag{7.74}$$

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

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

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

Figure 7.8b defines the final approximating preimage arrow. This arrow, the lifts in Figure 7.7, and the semantic function $[\![\cdot]\!]_{\mathsf{a}}$ in Figure 7.1 define an approximating semantics for partial, probabilistic programs.

7.9.3 Correctness

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

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

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

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

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

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

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

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

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

$$(\ggg'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows}' \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{pre}}{\rightrightarrows}' \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows}' \mathsf{Z})$$
$$(\mathsf{h}_1 \ggg'_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A} := \mathsf{let}\ \ \mathsf{h}'_1 := \mathsf{h}_1\ \mathsf{A}$$
$$\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{h}'_2 := \mathsf{h}_2\ (\mathsf{range}'_{\mathsf{pre}}\ \mathsf{h}'_1)$$
$$\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ \mathsf{h}'_2 \circ'_{\mathsf{pre}} \mathsf{h}'_1$$

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

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

$$\mathsf{lazy}'_{\mathsf{pre}} : (1 \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows}' \mathsf{Y})) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows}' \mathsf{Y})$$
$$\mathsf{lazy}'_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{A} := \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing'_{\mathsf{pre}}\ (\mathsf{h}\ 0\ \mathsf{A})$$

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

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

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

$$\mathsf{branch}'_{\mathsf{pre}^*} : \mathsf{X} \underset{\mathsf{pre}^*}{\rightrightarrows}' \mathsf{Bool}$$
$$\mathsf{branch}'_{\mathsf{pre}^*}\ \mathsf{j} :=$$
$$\quad \mathsf{fst}_{\mathsf{pre}} \ggg'_{\mathsf{pre}} \mathsf{snd}_{\mathsf{pre}} \ggg'_{\mathsf{pre}} \pi_{\mathsf{pre}}\ \mathsf{j}$$

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

$$\mathsf{ifte}^{\Downarrow'}_{\mathsf{pre}^*} : (\mathsf{X} \underset{\mathsf{pre}^*}{\rightrightarrows}' \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}^*}{\rightrightarrows}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}^*}{\rightrightarrows}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}^*}{\rightrightarrows}' \mathsf{Y})$$
$$\mathsf{ifte}^{\Downarrow'}_{\mathsf{pre}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} :=$$
$$\quad \mathsf{let}\ \ \langle \mathsf{C_k}, \mathsf{p_k} \rangle := \mathsf{k}_1\ (\mathsf{left\ j})\ \mathsf{A}$$
$$\qquad\quad \langle \mathsf{C_b}, \mathsf{p_b} \rangle := \mathsf{branch}_{\mathsf{pre}^*}\ \mathsf{j}\ \mathsf{A}$$
$$\qquad\qquad\quad \mathsf{C}_2 := \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{true}\}$$
$$\qquad\qquad\quad \mathsf{C}_3 := \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{false}\}$$
$$\qquad\qquad\quad \mathsf{A}_2 := \mathsf{p_k}\ \mathsf{C}_2 \cap \mathsf{p_b}\ \mathsf{C}_2$$
$$\qquad\qquad\quad \mathsf{A}_3 := \mathsf{p_k}\ \mathsf{C}_3 \cap \mathsf{p_b}\ \mathsf{C}_3$$
$$\qquad\ \mathsf{in}\ \ \mathsf{if}\ (\mathsf{C_b} = \{\mathsf{true}, \mathsf{false}\})$$
$$\qquad\qquad\quad \langle \top, \lambda \mathsf{B}.\, \mathsf{A}_2 \vee \mathsf{A}_3 \rangle$$
$$\qquad\qquad\quad (\mathsf{k}_2\ (\mathsf{left\ (right\ j)})\ \mathsf{A}_2 \uplus'_{\mathsf{pre}}\ \mathsf{k}_3\ (\mathsf{right\ (right\ j)})\ \mathsf{A}_3)$$

(c) Preimage* arrow combinators for probabilistic choice and guaranteed termination. Figure 7.6 (AStore arrow transformer) defines η'_{pre^*}, $(\ggg'_{\mathsf{pre}^*})$, $(\&\&\&'_{\mathsf{pre}^*})$, $\mathsf{ifte}'_{\mathsf{pre}^*}$ and $\mathsf{lazy}'_{\mathsf{pre}^*}$.

Figure 7.8: Implementable arrows that approximate preimage arrows. Specific lifts such as $\mathsf{fst}_{\mathsf{pre}} := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{fst}$ are computable (see Figure 7.7), but $\mathsf{arr}'_{\mathsf{pre}}$ is not.

Theorem 7.56 (sound)**.** *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle \langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle$ *and* $\mathsf{B} \in \mathsf{Rect\ Y}$, $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \subseteq \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$.

Proof. By construction. □

To use structural induction on the interpretation of e, we need a theorem that allows representing it as a finite expression (Definition 9.28). Because $\mathsf{ifte}^{\Downarrow'}_{\mathsf{pre}^*}$ does not branch when either branch could be taken, an equivalent finite expression exists for each rectangular domain subset A.

Theorem 7.57 (equivalent finite expression)**.** *Let* $\mathsf{A}' \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle$*. There is a finite expression* e' *for which* $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}''\ \mathsf{j}_0\ \mathsf{A}')\ \mathsf{B} = \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A}')\ \mathsf{B}$ *for all* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$*, where* $\mathsf{h}'' := [\![e']\!]^{\Downarrow'}_{\mathsf{pre}^*}$.

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

Corollary 7.58 (terminating)**.** *For all* $\mathsf{A}' \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$*,* $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A}')\ \mathsf{B}$ *terminates.*

Theorem 7.59 (monotone)**.** $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$ *is monotone in both* A *and* B*.*

Proof. Lattice operators $(\cap)$ and $(\vee)$ are monotone, as is $(\times)$. Therefore, $\mathsf{id}_{\mathsf{pre}}$ and the other lifts in Figure 7.7 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 7.8b) likewise preserve monotonicity, as does η'_{pre^*}; therefore $\mathsf{id}_{\mathsf{pre}^*}$ and other lifts are monotone.

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

For monotonicity in A, suppose $\mathsf{A}_1 \subseteq \mathsf{A}_2$. Apply Theorem 7.57 with $\mathsf{A}' := \mathsf{A}_1$ to yield e'; clearly, it is also an equivalent finite expression for A_2. Monotonicity follows from

structural induction on the interpretation of e'.

For monotonicity in B, use Theorem 7.57 with fixed $\mathsf{A}' := \mathsf{A}$. □

Theorem 7.60 (decreasing)**.** *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$, $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \subseteq \mathsf{A}$.

Proof. Because they compute exact preimages of rectangular sets under restriction to rectangular domains, $\mathsf{id}_{\mathsf{pre}}$ and the other lifts in Figure 7.7 are decreasing.

By definition and applying basic lattice properties,

$$\begin{aligned}
\mathsf{ap}'_{\mathsf{pre}}\ ((\mathsf{h}_1 \ggg'_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \mathsf{B}' \ \text{ for some } \mathsf{B}' \qquad (7.75)\\
\mathsf{ap}'_{\mathsf{pre}}\ ((\mathsf{h}_1 \mathbin{\&\&\&}'_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ (\mathsf{proj}_1\ \mathsf{B}) \cap \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_2\ \mathsf{A})\ (\mathsf{proj}_2\ \mathsf{B})\\
\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{ifte}'_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{h}_3\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{let}\ \ \mathsf{A}_2 := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \{\mathsf{true}\}\\
&\qquad\qquad\ \ \mathsf{A}_3 := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \{\mathsf{false}\}\\
&\qquad\qquad \mathsf{in}\ \ \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_2\ \mathsf{A}_2)\ \mathsf{B} \cup \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_3\ \mathsf{A}_3)\ \mathsf{B}\\
\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{lazy}'_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}\ 0\ \mathsf{A})\ \mathsf{B})
\end{aligned}$$

Thus, approximating preimage arrow combinators return decreasing computations when given decreasing computations. This property transfers trivially to approximating preimage* arrow combinators. Use Theorem 7.57 with $\mathsf{A}' := \mathsf{A}$, and structural induction. □

7.9.4 Preimage Refinement Algorithm

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

Definition 7.61 (preimage refinement algorithm)**.** *Let* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$ *and*

$$\begin{aligned}
&\mathsf{refine} : \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle \Rightarrow \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle\\
&\mathsf{refine}\ \mathsf{A} := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}
\end{aligned} \qquad (7.76)$$

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

rectangles, and define

$$\begin{aligned} &\mathsf{refine}^* : \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle) \Rightarrow \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle) \\ &\mathsf{refine}^*\ \mathcal{A} := \mathsf{image}\ \mathsf{refine}\ \left(\bigcup_{\mathsf{A}\in\mathcal{A}} \mathsf{partition}\ \mathsf{A}\right) \end{aligned} \tag{7.77}$$

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

Theorem 7.60 (decreasing) guarantees $\mathsf{refine}\ \mathsf{A}$ is never larger than A. Theorem 7.59 (monotone) guarantees refining a *partition* of A never does worse than refining A itself. Theorem 7.56 (sound) guarantees the algorithm is **sound**: the preimage of B is always contained in the covering partition refine^* returns.

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

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

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

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***.

7.10.1 Direct Implementations

If sets are restricted to be finite, the arrows used as translation targets in the exact semantics, defined in Figures. 4.1, 7.2, 7.3, 7.4, 7.5 and 7.6, 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.

Given a rectangular set library, the approximating preimage arrows defined in Figures. 7.7 and 7.8b can be implemented with few changes in any practical λ-calculus. We have done so in Typed Racket and Haskell [1]. Both implementations' arrow combinator definitions are almost line-for-line transliterations from the figures. They are at `https://github.com/ntoronto/writing/tree/master/2014esop-code`.

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

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

7.10.2 Dr. Bayes

Our main implementation, ***Dr. Bayes***, is written in Typed Racket. It consists of the semantic function $[\![\cdot]\!]_{\mathsf{a}^*}$ from Figure 7.1 and its extension $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a}^*}$, the bottom* arrow as defined in Figures. 7.2 and 7.6, the approximating preimage and preimage* arrows as defined in Figures. 7.7 and 7.8b, and algorithms to compute approximate probabilities. We use it to

test the feasibility of solving real-world problems by computing approximate preimages.

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

$$\begin{aligned}
\mathsf{mul}_{\mathsf{pre}} \;:=\; & \mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{fst}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{pos?}_{\mathsf{pre}}) \\
& \quad (\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{pos?}_{\mathsf{pre}}) \\
& \qquad \mathsf{mul}^{++}_{\mathsf{pre}} \\
& \qquad (\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{neg?}_{\mathsf{pre}}) \\
& \qquad\quad \mathsf{mul}^{+-}_{\mathsf{pre}} \\
& \qquad\quad (\mathsf{const}_{\mathsf{pre}}\ 0))) \\
& \quad \ldots
\end{aligned} \tag{7.78}$$

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

Section 7.9.4 outlines preimage refinement: a discretization algorithm that seems to converge for programs that halt with probability 1, consisting of repeatedly shrinking and repartitioning a program's domain. We do not use this algorithm directly in Dr. Bayes because it is inefficient. Good accuracy requires fine discretization, which is *exponential* in the number of discretized axes. For example, a nonrecursive program that contains only 10 uses of random would need to partition 10 axes of R, the set of random sources. Splitting each axis into only 4 disjoint intervals yields a partition of R of size $4^{10} = 1,048,576$.

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

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

one element in A, then

$$\mathsf{Pr}[\mathsf{B}'|\mathsf{B}] \;\approx\; \frac{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A}' \cap \mathsf{A})\ \mathsf{xs})}{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs})} \tag{7.79}$$

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

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

We cannot directly compute $\mathsf{a} \in \mathsf{A}$ or $\mathsf{a} \in \mathsf{A}' \cap \mathsf{A}$ in (7.79), but we can use the fact that A and A' are preimages, and use the interpretation of the program as a bottom arrow computation $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$:

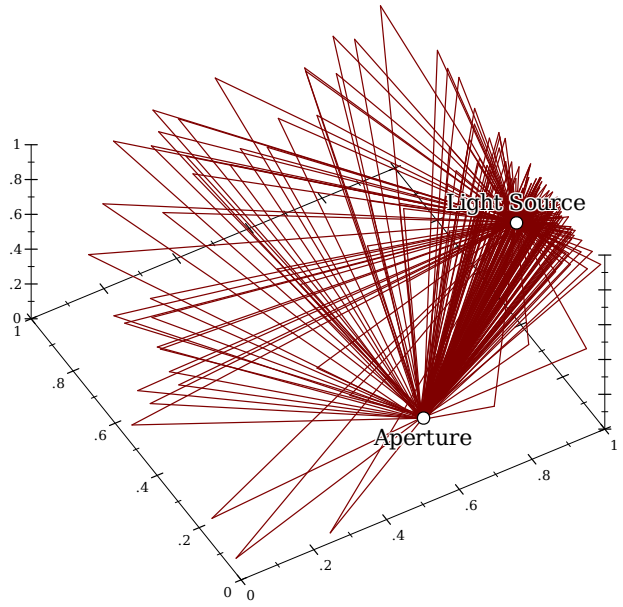
$$\begin{aligned}
\mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs} &= \mathsf{filter}\ (\in \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B})\ \mathsf{xs} \\
&= \mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{g}\ \mathsf{X}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs} \\
&= \mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs}
\end{aligned} \tag{7.80}$$

Substituting into (7.79) gives

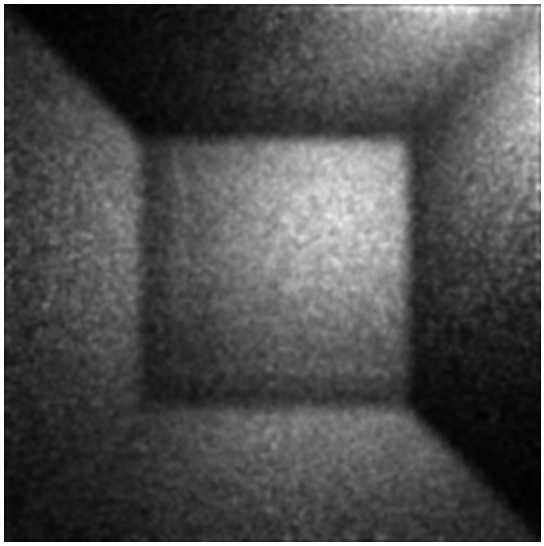
$$\mathsf{Pr}[\mathsf{B}'|\mathsf{B}] \;\approx\; \frac{\mathsf{length}\ (\mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B}' \cap \mathsf{B})\ \mathsf{xs})}{\mathsf{length}\ (\mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs})} \tag{7.81}$$

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

For simplicity, the preceeding discussion does not deal with projecting preimages from the domain of programs $(\mathsf{R} \times \mathsf{T}) \times \{\langle\rangle\}$ onto the set of random sources R. Shortly, Dr. Bayes samples rectangles from covering partitions of $(\mathsf{R} \times \mathsf{T}) \times \{\langle\rangle\}$ subsets, weights each rectangle by the inverse of the probability with which it is sampled, and projects onto



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



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

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

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

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

Figure 7.9: Stochastic ray tracing in Dr. Bayes is little more than physics simulation.

R. This alogorithm is a variant of **importance sampling** [15, Section 12.4], where the candidate distribution is defined by the sampling algorithm's partitioning choices, and the target distribution is P.

Figure 7.9 shows the result of using Dr. Bayes for stochastic ray tracing [54]. In this instance, photons are cast from a light source in a uniformly random direction and are reflected by the walls of a square room, generating paths. The objective is to sample, with the correct distribution, only those paths that pass through an aperture. The smaller the aperture, the smaller the probability a path passes through it, and the more focused the

resulting image.

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

7.11 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 [20], which are transformations from X to Y that can be run forward and backward, in a way that maintains some relationship between X and Y. Usually, a destructive, external process is assumed, so that, for example, a change from $\mathsf{y} \in \mathsf{Y}$ to $\mathsf{y}' \in \mathsf{Y}$ induces a corresponding change from $\mathsf{x} \in \mathsf{X}$ to some $\mathsf{x}' \in \mathsf{X}$. When transformations lose information, lenses must satisfy certain behavioral laws. In our work, no input or output is updated, and preimages are always definable regardless of non-injectivity.

Many multi-paradigm languages [19], especially constraint functional languages, bear a strong resemblance to our work. In fact, it is easy to add a fail expression to our semantics, or to transform constraints into boolean program outputs. The most obvious difference is evaluation strategy. The most important difference is that our interpretation of programs returns *distributions* of constrained outputs, rather than arbitrary single values that meet constraints.

The forward phase in computing preimages takes a subdomain and returns an over-approximation of the function's range for that subdomain. This clearly generalizes interval arithmetic [25] to all first-order algebraic types.

Our approximating semantics can be regarded as an abstract interpretation [13] where the concrete domain consists of measurable sets and the abstract domain consists

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

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

Some languages defined by an implementation are a probabilistic Scheme by Koller and Pfeffer [29], BUGS [34], BLOG [37], BLAISE [10], Church [18], and Kiselyov's embedded language for O'Caml based on continuations [27]. The reports on these languages generally describe interpreters, compilers, and algorithms for sampling with probabilistic conditions. Recently, Wingate et al [57, 58] have defined the semantics of *nonstandard interpretations* that enable efficient inference, but do not define the languages.

Early work in probabilistic language semantics is not motivated by Bayesian concerns, and thus does not address conditioning. Kozen [30] defines the meaning of bounded-space, imperative "while" programs as functions from probability measures to probability measures. Hurd [23] proves properties about programs with binary random choice by encoding programs and portions of measure theory in HOL. Jones [24] develops a domain-theoretic variation of probability theory, and with it defines the probability monad, whose discrete version is a distribution-valued variation of the set or list monad. Ramsey and Pfeffer [46] define the probability monad measure-theoretically and implement a language for finite probability. Park [42] extends a λ-calculus with probabilistic choice from a general class of probability measures using inverse transform sampling.

Some recent work in probabilistic language semantics tackles conditioning. Pfeffer's

IBAL [45] is the earliest lambda calculus with finite probabilistic choice that also defines conditional queries. Borgström et al [11] develop Fun, a first-order functional language without recursion, extended with probabilistic choice and conditioning. Its semantics interprets programs as *measure transformers* by transforming expressions into arrow-like combinators. The implementation generates a decomposition of the probability density represented by the program, if it exists. Bhat et al [9] replaces Fun's if with match, and interprets programs more directly as probability density functions by compositionally transforming expressions into an extension of the probability monad.

7.12 Conclusions and Future Work

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

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

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

Our semantics trades efficiency for simplicity by threading a constant, tree-shaped random source (Section 7.7.2). Passing subtrees instead would make random a constant-time primitive, and allow combinators to detect lack of change and return cached values. Other future optimization work includes creating new sampling algorithms, and using other easily measured but more expressive set representations, such as parallelotopes [4]. On the theory

side, we intend to explore preimage computation's connection to type checking and type inference, investigate ways to integrate and leverage polymorphic type systems, and find the conditions under which preimage refinement is complete in the limit.

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

Chapter 8

Implementation of Preimage Computation

8.1 Set Representation

8.2 Preimages Under Real Functions

XXX: lifts are generally uncomputable, but many specific lifts are computable or approximately computable

XXX: something about computing functions backward by computing inverses forward

XXX: obvious how to do this with invertible functions (though infinite endpoints are a little tricky); not obvious how to do the same with two-argument functions

8.2.1 Invertible Primitives

We consider only strictly monotone functions on $\mathbb{R}$. 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 8.1. *If* $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ *is strictly montone,* f *is injective. If* f *is additionally surjective,* f *and its inverse are continuous.*

Preimages under invertible functions can be computed using their inverses. We are primarily interested in computing preimages under restricted functions.

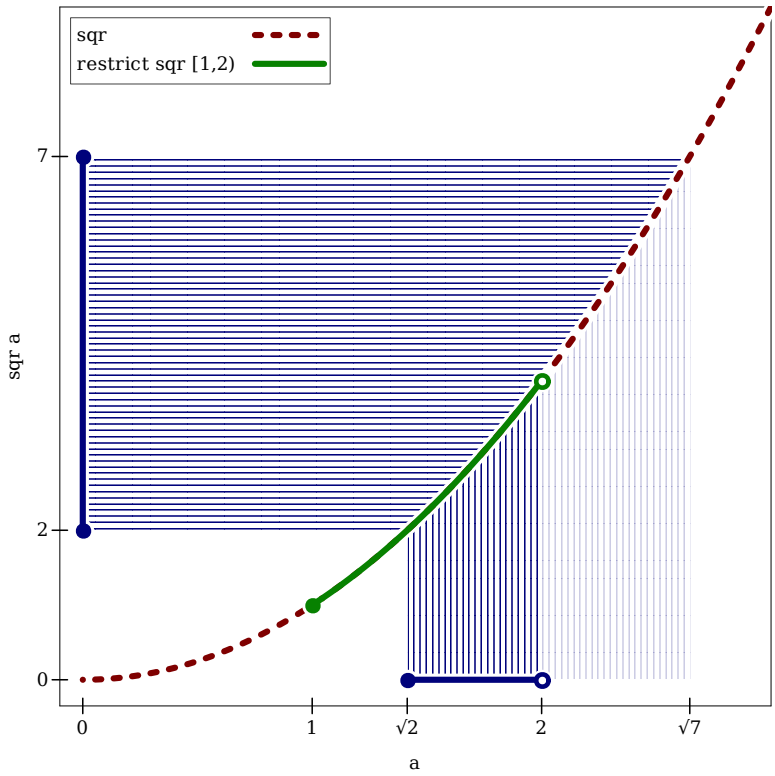


Figure 8.1: Computing the preimage of the interval $[2, 7]$ under sqr restricted to $[1, 2)$, by computing roots and intersecting with $[1, 2)$.

Lemma 8.2. *Let* $\mathsf{A}' \subseteq \mathsf{A}$, $\mathsf{B}' \subseteq \mathsf{B}$, *and* $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ *have inverse* $\mathsf{f}^{-1} : \mathsf{B} \to \mathsf{A}$. *Let* $\mathsf{f}' := \mathsf{restrict}\ \mathsf{f}\ \mathsf{A}'$. *Then*

$$\mathsf{preimage}\ \mathsf{f}'\ \mathsf{B}' \ = \ \mathsf{A}' \cap \mathsf{image}\ \mathsf{f}^{-1}\ \mathsf{B}' \tag{8.1}$$

These facts suggest that we can compute images (or preimages) of intervals under any strictly monotone, surjective f by applying f (or its inverse) to interval endpoints to yield an interval, as in Figure 8.1 This is evident for endpoints in A. Limit endpoints like $+\infty$ require a larger $\bar{\mathsf{f}}$ defined on a compact superset of A.

The next theorem is easier to state with interval notation in which the kind of interval is not baked into the syntax.

Definition 8.3 (interval)**.** $[\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!]$ *denotes an interval, where* $\mathsf{a}_1, \mathsf{a}_2 \in \overline{\mathbb{R}}$ *are extended real endpoints, and* $\alpha_1, \alpha_2 \in \mathsf{Bool}$ *determine whether* a_1 *and* a_2 *are contained in the interval.*

Example 8.4. Some intervals, using $[\![\cdot,\cdot,\cdot,\cdot]\!]$ notation:

$$\begin{aligned}
[\![0, 1, \mathsf{true}, \mathsf{false}]\!] &= [0, 1) \\
[\![-\infty, 0, \mathsf{false}, \mathsf{true}]\!] &= (-\infty, 0] \\
[\![-\infty, +\infty, \mathsf{false}, \mathsf{false}]\!] &= (-\infty, +\infty) = \mathbb{R} \\
[\![-\infty, +\infty, \mathsf{true}, \mathsf{true}]\!] &= [-\infty, +\infty] = \overline{\mathbb{R}}
\end{aligned} \tag{8.2}$$

All but the last are subsets of $\mathbb{R}$. ♢

Theorem 8.5 (images of intervals by endpoints)**.** *Let* $\overline{\mathsf{A}}$ *and* $\overline{\mathsf{B}}$ *be compact subsets of* $\overline{\mathbb{R}}$*,* $\bar{\mathsf{f}} : \overline{\mathsf{A}} \to \overline{\mathsf{B}}$ *strictly monotone and surjective, and* f *the restriction of* $\bar{\mathsf{f}}$ *to some* $\mathsf{A} \subseteq \overline{\mathsf{A}}$*. For all nonempty* $[\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!] \subseteq \mathsf{A}$*,*

- *If* $\bar{\mathsf{f}}$ *is increasing,* $\mathsf{image}\ \mathsf{f}\ [\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!] = [\![\bar{\mathsf{f}}\ \mathsf{a}_1, \bar{\mathsf{f}}\ \mathsf{a}_2, \alpha_1, \alpha_2]\!]$.
- *If* $\bar{\mathsf{f}}$ *is decreasing,* $\mathsf{image}\ \mathsf{f}\ [\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!] = [\![\bar{\mathsf{f}}\ \mathsf{a}_2, \bar{\mathsf{f}}\ \mathsf{a}_1, \alpha_2, \alpha_1]\!]$.

Proof. Because $\overline{\mathsf{A}}$ is compact and totally ordered, every subset of $\overline{\mathsf{A}}$ has a lower and an upper bound in $\overline{\mathsf{A}}$. Therefore, the endpoints of every interval subset of A are in $\overline{\mathsf{A}}$.

Let $(\mathsf{a}_1, \mathsf{a}_2] \subseteq \mathsf{A}$. Suppose $\bar{\mathsf{f}}$ is strictly increasing; thus $\mathsf{a}_1 < \mathsf{a} \leq \mathsf{a}_2$ if and only if $\bar{\mathsf{f}}\ \mathsf{a}_1 < \bar{\mathsf{f}}\ \mathsf{a} \leq \bar{\mathsf{f}}\ \mathsf{a}_2$, so $\mathsf{image}\ \mathsf{f}\ (\mathsf{a}_1, \mathsf{a}_2] = \mathsf{image}\ \bar{\mathsf{f}}\ (\mathsf{a}_1, \mathsf{a}_2] = (\bar{\mathsf{f}}\ \mathsf{a}_1, \bar{\mathsf{f}}\ \mathsf{a}_2]$. The remaining cases are similar. □

To use Theorem 8.5 to compute preimages under f by computing images under its inverse f^{-1}, we must know whether f^{-1} is increasing or decreasing. The following lemma can help.

Lemma 8.6. *If* $\mathsf{f} : \mathsf{A} \to \mathsf{B}$ *is strictly monotone and surjective with inverse* $\mathsf{f}^{-1} : \mathsf{B} \to \mathsf{A}$*, then* f *is increasing if and only if* f^{-1} *is increasing.*

Example 8.7. The extension of $\mathsf{log} : (0, +\infty) \to \mathbb{R}$ to the compact superdomain $[0, +\infty]$ is

defined by

$$\begin{aligned}
&\overline{\mathsf{log}} : [0, +\infty] \to \overline{\mathbb{R}} \\
&\overline{\mathsf{log}}\ \mathsf{a} := \lim_{\mathsf{a}' \to \mathsf{a}} \mathsf{log}\ \mathsf{a}' = \mathsf{case}\ \ \mathsf{a} \\
&\qquad\qquad\qquad\qquad\qquad\quad 0 \quad\longrightarrow\ -\infty \\
&\qquad\qquad\qquad\qquad\qquad\quad +\infty \ \longrightarrow\ +\infty \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else} \ \longrightarrow\ \mathsf{log}\ \mathsf{a}
\end{aligned} \tag{8.3}$$

The extension of its inverse exp is $\overline{\mathsf{exp}} : \overline{\mathbb{R}} \to [0, +\infty]$, defined similarly, which by Lemma 8.6 is also strictly increasing. Thus,

$$\begin{aligned}
\mathsf{image}\ \mathsf{log}\ (0, 1] &= (\overline{\mathsf{log}}\ 0, \overline{\mathsf{log}}\ 1] = (-\infty, 0] \\
\mathsf{preimage}\ \mathsf{log}\ [0, +\infty) &= \mathsf{image}\ \mathsf{exp}\ [0, +\infty) \\
&= [\overline{\mathsf{exp}}\ 0, \overline{\mathsf{exp}}\ +\infty) = [1, +\infty)
\end{aligned} \tag{8.4}$$

by Theorem 8.5 and Lemma 8.2, ♢

8.2.2 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 8.8 (axial inverse)**.** *Let* $\mathsf{f_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$*. Functions* $\mathsf{f_a} : \mathsf{B} \times \mathsf{C} \to \mathsf{A}$ *and* $\mathsf{f_b} : \mathsf{C} \times \mathsf{A} \to \mathsf{B}$ *defined so that*

$$\mathsf{f_c}\ \langle \mathsf{a}, \mathsf{b} \rangle = \mathsf{c} \iff \mathsf{f_a}\ \langle \mathsf{b}, \mathsf{c} \rangle = \mathsf{a} \iff \mathsf{f_b}\ \langle \mathsf{c}, \mathsf{a} \rangle = \mathsf{b} \tag{8.5}$$

are ***axial inverses*** *with respect to* $\mathsf{f_c}$*'s first and second arguments.*

We call $\mathsf{f_c}$ ***axis-invertible*** or ***trijective*** when it has axial inverses $\mathsf{f_a}$ and $\mathsf{f_b}$. We call $\mathsf{f_a}$ the ***first axial inverse*** of $\mathsf{f_c}$ because it is the inverse of $\mathsf{f_c}$ along the first axis: $\mathsf{f_a}$ with only

c varying (i.e. $\lambda \mathsf{c} \in \mathsf{C}.\, \mathsf{f}_\mathsf{a}\ \langle \mathsf{b}, \mathsf{c} \rangle$), is the inverse of f_c with only a varying (i.e. $\lambda \mathsf{a} \in \mathsf{A}.\, \mathsf{f}_\mathsf{c}\ \langle \mathsf{a}, \mathsf{b} \rangle$). Similarly, f_b is the ***second axial inverse***.

Example 8.9. Let $\mathsf{add}_\mathsf{c} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, $\mathsf{add}_\mathsf{c}\ \langle \mathsf{a}, \mathsf{b} \rangle := \mathsf{a} + \mathsf{b}$. Its axial inverses are $\mathsf{add}_\mathsf{a}\ \langle \mathsf{b}, \mathsf{c} \rangle := \mathsf{c} - \mathsf{b}$ and $\mathsf{add}_\mathsf{b}\ \langle \mathsf{c}, \mathsf{a} \rangle := \mathsf{c} - \mathsf{a}$. ♢

We have chosen the axial inverse function types carefully: they are the only types for which f_c, f_a and f_b form a cyclic group.

Lemma 8.10. *The following statements are equivalent.*

- f_c *has axial inverses* f_a *and* f_b.
- f_a *has axial inverses* f_b *and* f_c.
- f_b *has axial inverses* f_c *and* f_a.

Equivalently, every axis-invertible function generates a cyclic group of order 3 by inversion in the first axis.

This fact is analogous to how mutual inverses f and f^{-1} also form a cyclic group (of order 2, generated by inversion). Similar to using mutual inversion to compute preimages under both log and exp, Lemma 8.10 allows computing preimages under two-argument functions related by axial inversion.

Example 8.11. Define $\mathsf{sub}_\mathsf{c} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ by $\mathsf{sub}_\mathsf{c}\ \langle \mathsf{a}, \mathsf{b} \rangle := \mathsf{a} - \mathsf{b}$. Because $\mathsf{sub}_\mathsf{c} = \mathsf{add}_\mathsf{b}$, $\mathsf{sub}_\mathsf{a} = \mathsf{add}_\mathsf{c}$ and $\mathsf{sub}_\mathsf{b} = \mathsf{add}_\mathsf{a}$. ♢

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 8.12 (preimage bounds from axial inverse images)**.** *Let* $\mathsf{A}' \subseteq \mathsf{A}$, $\mathsf{B}' \subseteq \mathsf{B}$, $\mathsf{C}' \subseteq \mathsf{C}$, *and* $\mathsf{f}_\mathsf{c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ *with axial inverses* f_a *and* f_b. *If* $\mathsf{f}'_\mathsf{c} = \mathsf{restrict}\ \mathsf{f}_\mathsf{c}\ (\mathsf{A}' \times \mathsf{B}')$, *then*

$$\begin{aligned} \mathsf{preimage}\ \mathsf{f}'_\mathsf{c}\ \mathsf{C}' \ \subseteq\ & (\mathsf{A}' \cap \mathsf{image}\ \mathsf{f}_\mathsf{a}\ (\mathsf{B}' \times \mathsf{C}')) \times \\ & (\mathsf{B}' \cap \mathsf{image}\ \mathsf{f}_\mathsf{b}\ (\mathsf{C}' \times \mathsf{A}')) \end{aligned} \tag{8.6}$$

Further, the right-hand side is the smallest rectangular superset.

Proof. The smallest rectangle containing $\mathsf{preimage}\ \mathsf{f}'_\mathsf{c}\ \mathsf{C}'$ is

$$\mathsf{preimage}\ \mathsf{f}'_\mathsf{c}\ \mathsf{C}' \subseteq (\mathsf{image}\ \mathsf{fst}\ (\mathsf{preimage}\ \mathsf{f}'_\mathsf{c}\ \mathsf{C}')) \times (\mathsf{image}\ \mathsf{snd}\ (\mathsf{preimage}\ \mathsf{f}'_\mathsf{c}\ \mathsf{C}')) \tag{8.7}$$

Starting with the first set in the product, expand definitions, distribute fst, replace $\mathsf{f_c}\ \langle \mathsf{a}, \mathsf{b}\rangle = \mathsf{c}$ by $\mathsf{f_a}\ \langle \mathsf{b}, \mathsf{c}\rangle = \mathsf{a}$, and simplify:

$$\begin{aligned}
&\mathsf{image}\ \mathsf{fst}\ (\mathsf{preimage}\ \mathsf{f}'_\mathsf{c}\ \mathsf{C}') \\
&= \mathsf{image}\ \mathsf{fst}\ \{\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{A}' \times \mathsf{B}' \mid \mathsf{f_c}\ \langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{C}'\} \\
&= \{\mathsf{a} \in \mathsf{A}' \mid \exists\, \mathsf{b} \in \mathsf{B}'.\, \mathsf{f_c}\ \langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{C}'\} \\
&= \{\mathsf{a} \in \mathsf{A}' \mid \exists\, \mathsf{b} \in \mathsf{B}', \mathsf{c} \in \mathsf{C}'.\, \mathsf{f_c}\ \langle \mathsf{a}, \mathsf{b}\rangle = \mathsf{c}\} \\
&= \{\mathsf{a} \in \mathsf{A}' \mid \exists\, \mathsf{b} \in \mathsf{B}', \mathsf{c} \in \mathsf{C}'.\, \mathsf{f_a}\ \langle \mathsf{b}, \mathsf{c}\rangle = \mathsf{a}\} \\
&= \{\mathsf{f_a}\ \langle \mathsf{b}, \mathsf{c}\rangle \mid \mathsf{b} \in \mathsf{B}', \mathsf{c} \in \mathsf{C}', \mathsf{f_a}\ \langle \mathsf{b}, \mathsf{c}\rangle \in \mathsf{A}'\} \\
&= \mathsf{A}' \cap \{\mathsf{f_a}\ \langle \mathsf{b}, \mathsf{c}\rangle \mid \mathsf{b} \in \mathsf{B}', \mathsf{c} \in \mathsf{C}'\} \\
&= \mathsf{A}' \cap \mathsf{image}\ \mathsf{f_a}\ (\mathsf{B}' \times \mathsf{C}')
\end{aligned}$$

The second set in the product is similar. □

Example 8.13. See Figure 8.2. Let $\mathsf{add}'_\mathsf{c} := \mathsf{restrict}\ \mathsf{add_c}\ ([0,1] \times [0,2])$. By Theorem 8.12,

$$\begin{aligned}
\mathsf{preimage}\ \mathsf{add}'_\mathsf{c}\ [0, \tfrac{1}{2}] &\subseteq ([0,1] \cap \mathsf{image}\ \mathsf{add_a}\ ([0,2] \times [0, \tfrac{1}{2}])) \times \\
&\quad\ ([0,2] \cap \mathsf{image}\ \mathsf{add_b}\ ([0, \tfrac{1}{2}] \times [0,1])) \\
&= ([0,1] \cap [-2, \tfrac{1}{2}]) \times ([0,2] \cap [-1, \tfrac{1}{2}]) \\
&= [0, \tfrac{1}{2}] \times [0, \tfrac{1}{2}]
\end{aligned}$$

is the smallest rectangular subset of $[0,1] \times [0,2]$ that contains the preimage of $[0, \frac{1}{2}]$ under $\mathsf{add_c}$. ♢

At this point, we have an analogue of Lemma 8.2, in that we can compute (approximate) preimages by computing images under (axial) inverses. To compute images using interval

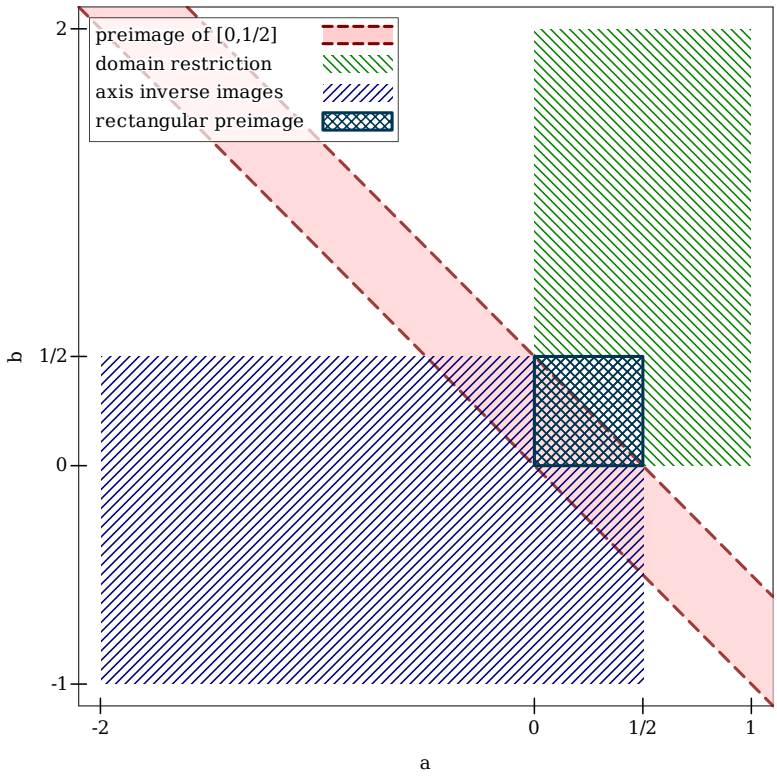


Figure 8.2: Computing an approximate preimage of $[0, \frac{1}{2}]$ under addition restricted to $[0, 1] \times [0, 2]$ (Example 8.13). The preimage is approximated by intersecting the domain with an overapproximation computed using axial inverses.

endpoints, we need analogues of Lemma 8.1 (strictly monotone, surjective functions are invertible and continuous), Theorem 8.5 (images of intervals by endpoints), and Lemma 8.6 (inverse direction).

We first need a notion of properties that hold along an axis for every fixed value of the other argument.

Definition 8.14 (uniform axis property)**.** $\mathsf{f_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ *has property* P ***uniformly*** *in its first axis when* $\mathsf{P}\ (\mathsf{flip}\ (\mathsf{curry}\ \mathsf{f_c})\ \mathsf{b})$ *for all* $\mathsf{b} \in \mathsf{B}$*, and uniformly in its second axis when* $\mathsf{P}\ (\mathsf{curry}\ \mathsf{f_c}\ \mathsf{a})$ *for all* $\mathsf{a} \in \mathsf{A}$*. If the axis is not specified,* P *holds uniformly for both.*

Now Lemma 8.1's analogue is an easy corollary.

Lemma 8.15. *Let* $\mathsf{f_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ *for totally ordered* A*,* B *and* C*. If* $\mathsf{f_c}$ *is uniformly surjective and either uniformly strictly increasing or uniformly strictly decreasing in each axis, then* $\mathsf{f_c}$

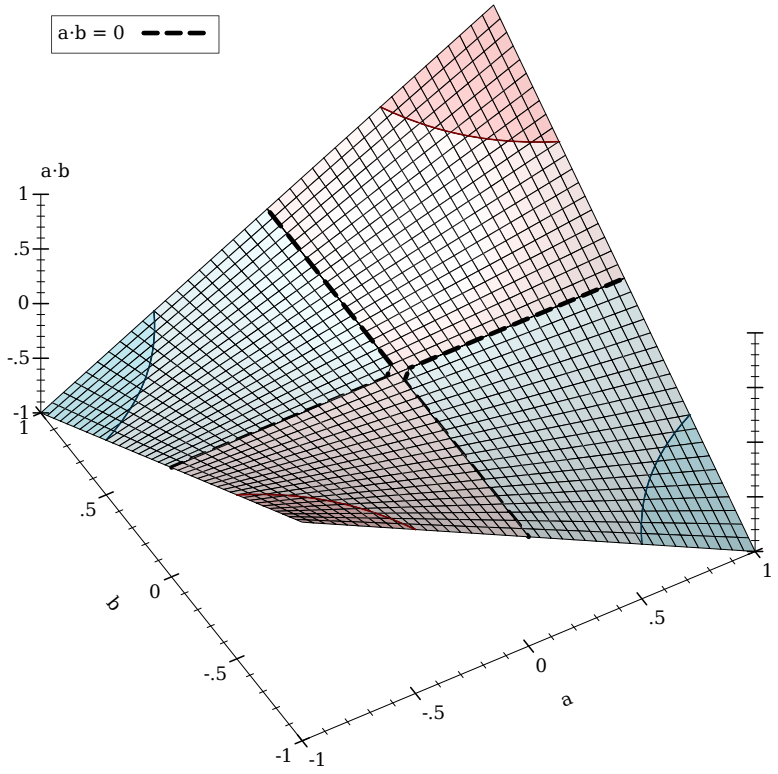


Figure 8.3: Multiplication on $\mathbb{R} \times \mathbb{R}$ is not uniformly surjective, nor uniformly strictly increasing or decreasing in each axis: $\mathsf{a} \cdot 0 = 0$ and $0 \cdot \mathsf{b} = 0$ for all a and b (Example 8.18). Fortunately, it is uniformly surjective and strictly monotone in each open quadrant.

is axis-invertible; further, it and its axial inverses are continuous.

From here on, assume axis monotonicity properties are uniform unless otherwise stated.

Example 8.16. $\mathsf{add_c}$ is uniformly surjective and strictly increasing. $\mathsf{sub_c}$ is uniformly surjective and strictly increasing/decreasing in its first/second axis. Therefore, both are axis-invertible. ♢

Restriction usually makes a function not uniformly surjective.

Example 8.17. Let $\mathsf{add}'_\mathsf{c} : [0,1] \times [0,1] \to [0,2]$, defined by restricting $\mathsf{add_c}$. It is strictly increasing, but not uniformly surjective: the range of $\mathsf{curry}\ \mathsf{add}'_\mathsf{c}\ 0$ is $[0,1]$, not $[0,2]$. ♢

Fortunately, restriction sometimes does the opposite.

Example 8.18. Define $\mathsf{mul_c} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ by $\mathsf{mul_c}\ \langle \mathsf{a}, \mathsf{b}\rangle := \mathsf{a} \cdot \mathsf{b}$. It is not uniformly surjective nor strictly monotone because $\mathsf{mul_c}\ \langle 0, \mathsf{b}\rangle = 0$ for all $\mathsf{b} \in \mathsf{B}$. (See Figure 8.3.) But $\mathsf{mul_c^{++}} : (0, +\infty) \times (0, +\infty) \to (0, +\infty)$, and $\mathsf{mul_c}$ restricted to the other open quadrants, are uniformly surjective and strictly increasing or decreasing in each axis. ♢

Theorem 8.5 justifies computing images of intervals with infinite endpoints 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 8.19. $\mathsf{add_c}$ cannot be extended to $\overline{\mathsf{add_c}} : \overline{\mathbb{R}} \times \overline{\mathbb{R}} \to \overline{\mathbb{R}}$ in the same way log is extended to $\overline{\mathsf{log}}$ because

$$\lim_{\langle \mathsf{a}', \mathsf{b}'\rangle \to \langle \mathsf{a}, \mathsf{b}\rangle} \mathsf{add_c}\ \langle \mathsf{a}', \mathsf{b}'\rangle \tag{8.8}$$

does not exist when $\langle \mathsf{a}, \mathsf{b}\rangle$ is $\langle -\infty, +\infty\rangle$ or $\langle +\infty, -\infty\rangle$. ♢

The previous example suggests that extensions of 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 8.20 ($\overline{\mathbb{R}} \times \overline{\mathbb{R}}$ extension)**.** *Let* A, B, C *be open subsets of* $\mathbb{R}$*, with* $\mathsf{f_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ *uniformly surjective and strictly increasing or decreasing in each axis. Let* $\overline{\mathsf{A}}$, $\overline{\mathsf{B}}$ *and* $\overline{\mathsf{C}}$ *be the closures of* A, B *and* C *in* $\overline{\mathbb{R}}$*. The following extension is well-defined:*

$$\begin{aligned} &\overline{\mathsf{f_c}} : (\overline{\mathsf{A}} \times \overline{\mathsf{B}}) \backslash \mathsf{N} \to \overline{\mathsf{C}} \\ &\overline{\mathsf{f_c}}\ \langle \mathsf{a}, \mathsf{b}\rangle \ := \lim_{\langle \mathsf{a}', \mathsf{b}'\rangle \to \langle \mathsf{a}, \mathsf{b}\rangle} \mathsf{f_c}\ \langle \mathsf{a}', \mathsf{b}'\rangle \end{aligned} \tag{8.9}$$

where $\mathsf{N} := \{\langle \min\ \overline{\mathsf{A}}, \max\ \overline{\mathsf{B}}\rangle, \langle \max\ \overline{\mathsf{A}}, \min\ \overline{\mathsf{B}}\rangle\}$ *if* $\mathsf{f_c}$ *is increasing or decreasing, and* $\mathsf{N} := \{\langle \min\ \overline{\mathsf{A}}, \min\ \overline{\mathsf{B}}\rangle, \langle \max\ \overline{\mathsf{A}}, \max\ \overline{\mathsf{B}}\rangle\}$ *if* $\mathsf{f_c}$ *is increasing/decreasing or decreasing/increasing.*

Proof. Suppose $\mathsf{f_c}$ is increasing, and let $\mathsf{g} : \mathbb{N} \to \mathsf{A} \times \mathsf{B}$ be a sequence of $\mathsf{f_c}$'s domain values.

Any g that converges to $\langle \max\ \overline{\mathsf{A}}, \max\ \overline{\mathsf{B}}\rangle$ has a strictly increasing subsequence. By monotonicity, $\mathsf{map}\ \mathsf{f_c}\ \mathsf{g}$ has a strictly increasing subsequence. It is bounded by $\max\ \overline{\mathsf{C}}$, so $\overline{\mathsf{f_c}}\ \langle \max\ \overline{\mathsf{A}}, \max\ \overline{\mathsf{B}}\rangle = \max\ \overline{\mathsf{C}}$. A similar argument proves $\overline{\mathsf{f}}\ \langle \min\ \overline{\mathsf{A}}, \min\ \overline{\mathsf{B}}\rangle = \min\ \overline{\mathsf{C}}$.

For any g that converges to $\langle \mathsf{max}\ \overline{\mathsf{A}}, \mathsf{b}' \rangle$ for some $\mathsf{b}' \in \mathsf{B}$, define

$$\mathsf{g}' \ := \ \mathsf{map}\ (\lambda \langle \mathsf{a}, \mathsf{b} \rangle .\, \langle \mathsf{f_a}\ \langle \mathsf{b}', \mathsf{f_c}\ \langle \mathsf{a}, \mathsf{b} \rangle \rangle', \mathsf{b}' \rangle)\ \mathsf{g} \tag{8.10}$$

where $\mathsf{f_a}$ is $\mathsf{f_c}$'s first axial inverse, so that $\mathsf{map}\ \mathsf{f_c}\ \mathsf{g} = \mathsf{map}\ \mathsf{f_c}\ \mathsf{g}'$. Because g' 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, $\mathsf{map}\ \mathsf{f_c}\ \mathsf{g}'$ has a strictly increasing subsequence. It is bounded by $\mathsf{max}\ \overline{\mathsf{C}}$, so $\overline{\mathsf{f_c}}\ \langle \mathsf{max}\ \overline{\mathsf{A}}, \mathsf{b}' \rangle = \mathsf{max}\ \overline{\mathsf{C}}$. By similar arguments, $\overline{\mathsf{f}}\ \langle \mathsf{min}\ \overline{\mathsf{A}}, \mathsf{b} \rangle = \mathsf{min}\ \overline{\mathsf{C}}$ and so on.

Arguments for f decreasing, etc., are similar. □

Following the proof of Theorem 8.20, extensions of two-argument functions can be defined by two corner cases, four border cases, and an interior case.

Example 8.21. Define $\mathsf{pow_c} : (0,1) \times (0,+\infty) \to (0,1)$ by $\mathsf{pow_c}\ \langle \mathsf{a}, \mathsf{b} \rangle := \mathsf{exp}\ (\mathsf{b} \cdot \mathsf{log}\ \mathsf{a})$, which is increasing/decreasing. Its extension to a subset of $\overline{\mathbb{R}} \times \overline{\mathbb{R}}$ is

$$\begin{aligned}
&\overline{\mathsf{pow_c}} : ([0,1] \times [0,+\infty]) \backslash \mathsf{N} \to [0,1] \\
&\overline{\mathsf{pow_c}}\ \langle \mathsf{a}, \mathsf{b} \rangle \ := \ \mathsf{case}\ \ \langle \mathsf{a}, \mathsf{b} \rangle \\
&\qquad\qquad\qquad\qquad \langle 0, +\infty \rangle \ \longrightarrow\ 0 \\
&\qquad\qquad\qquad\qquad \langle 1, 0 \rangle \ \longrightarrow\ 1 \\
&\qquad\qquad\qquad\qquad \langle 0, \mathsf{b} \rangle \ \longrightarrow\ 0 \\
&\qquad\qquad\qquad\qquad \langle 1, \mathsf{b} \rangle \ \longrightarrow\ 1 \\
&\qquad\qquad\qquad\qquad \langle \mathsf{a}, 0 \rangle \ \longrightarrow\ 1 \\
&\qquad\qquad\qquad\qquad \langle \mathsf{a}, +\infty \rangle \ \longrightarrow\ 0 \\
&\qquad\qquad\qquad\qquad \mathsf{else} \ \longrightarrow\ \mathsf{pow_c}\ \langle \mathsf{a}, \mathsf{b} \rangle
\end{aligned} \tag{8.11}$$

where $\mathsf{N} := \{\langle 0, 0 \rangle, \langle 1, +\infty \rangle\}$. ♢

The analogue of Theorem 8.5 is easiest to state if we have predicates that indicate a function's direction in each axis. Define $\mathsf{inc_1} : (\mathsf{A} \times \mathsf{B} \to \mathsf{C}) \Rightarrow \mathsf{Bool}$ so that $\mathsf{inc_1}\ \mathsf{f}$ if and only if f is strictly increasing in its first axis, and similarly $\mathsf{inc_2}$ so that $\mathsf{inc_2}\ \mathsf{f}$ if and only if f is strictly increasing in its second axis.

Theorem 8.22 (images of rectangles by interval endpoints)**.** *Let* $\mathsf{A}, \mathsf{B}, \mathsf{C}$ *be open subsets of* $\mathbb{R}$*, with* $\mathsf{f_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ *uniformly surjective and strictly increasing or decreasing in*

each axis, with extension $\overline{\mathsf{f_c}}$ *as defined in Theorem 8.20. If* $\mathsf{A}' := [\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!] \subseteq \mathsf{A}$ *and* $\mathsf{B}' := [\![\mathsf{b}_1, \mathsf{b}_2, \beta_1, \beta_2]\!] \subseteq \mathsf{B}$*, then the image* C' *under* $\mathsf{f_c}$ *is*

$$\begin{aligned}
&\mathsf{image}\ \mathsf{f_c}\ ([\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!] \times [\![\mathsf{b}_1, \mathsf{b}_2, \beta_1, \beta_2]\!]) \\
&= \mathsf{let}\ \ \langle \mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2 \rangle := \mathsf{cond}\ \ (\mathsf{inc}_1\ \mathsf{f_c}) \longrightarrow \langle \mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2 \rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else} \longrightarrow \langle \mathsf{a}_2, \mathsf{a}_1, \alpha_2, \alpha_1 \rangle \\
&\qquad\qquad \langle \mathsf{b}_1, \mathsf{b}_2, \beta_1, \beta_2 \rangle := \mathsf{cond}\ \ (\mathsf{inc}_2\ \mathsf{f_c}) \longrightarrow \langle \mathsf{b}_1, \mathsf{b}_2, \beta_1, \beta_2 \rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else} \longrightarrow \langle \mathsf{b}_2, \mathsf{b}_1, \beta_2, \beta_1 \rangle \\
&\qquad \mathsf{in}\ \ [\![\overline{\mathsf{f_c}}\ \langle \mathsf{a}_1, \mathsf{b}_1 \rangle, \overline{\mathsf{f_c}}\ \langle \mathsf{a}_2, \mathsf{b}_2 \rangle, \alpha_1\ \mathsf{and}\ \beta_1, \alpha_2\ \mathsf{and}\ \beta_2]\!]
\end{aligned} \tag{8.12}$$

Proof. Because $\mathsf{f_c}$ is continuous and $\mathsf{A}' \times \mathsf{B}'$ is a connected set, C' is a connected set, which in $\mathbb{R}$ is an interval. Thus, we need to determine only its bounds and whether it contains each endpoint.

Suppose $\mathsf{f_c}$ is increasing. By monotonicity, C' is contained in $[\overline{\mathsf{f_c}}\ \langle \mathsf{a}_1, \mathsf{b}_1 \rangle, \overline{\mathsf{f_c}}\ \langle \mathsf{a}_2, \mathsf{b}_2 \rangle]$. If either α_1 or β_1 is false, C' cannot contain $\overline{\mathsf{f_c}}\ \langle \mathsf{a}_1, \mathsf{b}_1 \rangle$. If either α_2 or β_2 is false, C' cannot contain $\overline{\mathsf{f_c}}\ \langle \mathsf{a}_2, \mathsf{b}_2 \rangle$. Therefore $\mathsf{C}' = [\![\overline{\mathsf{f_c}}\ \langle \mathsf{a}_1, \mathsf{b}_1 \rangle, \overline{\mathsf{f_c}}\ \langle \mathsf{a}_2, \mathsf{b}_2 \rangle, \alpha_1\ \mathsf{and}\ \beta_1, \alpha_2\ \mathsf{and}\ \beta_2]\!]$.

We still must prove $\langle \mathsf{a}_1, \mathsf{b}_1 \rangle$ and $\langle \mathsf{a}_2, \mathsf{b}_2 \rangle$ are in $\overline{\mathsf{f_c}}$'s domain. First, recall $\overline{\mathsf{f_c}} : (\overline{\mathsf{A}} \times \overline{\mathsf{B}}) \backslash \mathsf{N} \to \overline{\mathsf{C}}$, where $\overline{\mathsf{A}}$, $\overline{\mathsf{B}}$ and $\overline{\mathsf{C}}$ are the closures of A, B and C in $\overline{\mathbb{R}}$, and $\mathsf{N} = \{\langle \mathsf{min}\ \overline{\mathsf{A}}, \mathsf{max}\ \overline{\mathsf{B}} \rangle, \langle \mathsf{max}\ \overline{\mathsf{A}}, \mathsf{min}\ \overline{\mathsf{B}} \rangle\}$.

Because $\mathsf{A}' \subseteq \mathsf{A}$ and $\mathsf{B}' \subseteq \mathsf{B}$, and A and B are open sets, $\mathsf{a}_1 \neq \mathsf{max}\ \overline{\mathsf{A}}$, $\mathsf{a}_2 \neq \mathsf{min}\ \overline{\mathsf{A}}$, $\mathsf{b}_1 \neq \mathsf{max}\ \overline{\mathsf{B}}$, and $\mathsf{b}_2 \neq \mathsf{min}\ \overline{\mathsf{B}}$, so

$$\begin{aligned}
\langle \mathsf{a}_1, \mathsf{b}_1 \rangle &\neq \langle \mathsf{max}\ \overline{\mathsf{A}}, \mathsf{b} \rangle \quad \text{for all } \mathsf{b} \in \overline{\mathsf{B}} \\
\langle \mathsf{a}_1, \mathsf{b}_1 \rangle &\neq \langle \mathsf{a}, \mathsf{max}\ \overline{\mathsf{B}} \rangle \quad \text{for all } \mathsf{a} \in \overline{\mathsf{A}} \\
\langle \mathsf{a}_2, \mathsf{b}_2 \rangle &\neq \langle \mathsf{min}\ \overline{\mathsf{A}}, \mathsf{b} \rangle \quad \text{for all } \mathsf{b} \in \overline{\mathsf{B}} \\
\langle \mathsf{a}_2, \mathsf{b}_2 \rangle &\neq \langle \mathsf{a}, \mathsf{min}\ \overline{\mathsf{B}} \rangle \quad \text{for all } \mathsf{a} \in \overline{\mathsf{A}}
\end{aligned} \tag{8.13}$$

Therefore, $\langle \mathsf{a}_1, \mathsf{b}_1 \rangle \notin \mathsf{N}$ and $\langle \mathsf{a}_2, \mathsf{b}_2 \rangle \notin \mathsf{N}$, as desired.

The remaining cases for $\mathsf{f_c}$ are similar. □

Example 8.23. Because $\mathsf{inc}_1\ \mathsf{pow_c}$ and $\mathsf{not}\ (\mathsf{inc}_2\ \mathsf{pow_c})$,

$$\begin{aligned}
&\mathsf{image}\ \mathsf{pow_c}\ ((0, \tfrac{1}{2}] \times [2, +\infty)) \\
&= \mathsf{let}\ \ \langle \mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2 \rangle := \langle 0, \tfrac{1}{2}, \mathsf{false}, \mathsf{true} \rangle \\
&\qquad\quad \langle \mathsf{b}_1, \mathsf{b}_2, \beta_1, \beta_2 \rangle := \langle +\infty, 2, \mathsf{false}, \mathsf{true} \rangle \\
&\qquad \mathsf{in}\ \ [\![\overline{\mathsf{pow_c}}\ \langle \mathsf{a}_1, \mathsf{b}_1 \rangle, \overline{\mathsf{pow_c}}\ \langle \mathsf{a}_2, \mathsf{b}_2 \rangle, \alpha_1\ \mathsf{and}\ \beta_1, \alpha_2\ \mathsf{and}\ \beta_2]\!] \\
&= [\![\overline{\mathsf{pow_c}}\ \langle 0, +\infty \rangle, \overline{\mathsf{pow_c}}\ \langle \tfrac{1}{2}, 2 \rangle, \mathsf{false}\ \mathsf{and}\ \mathsf{false}, \mathsf{true}\ \mathsf{and}\ \mathsf{true}]\!] \\
&= [\![0, \tfrac{1}{4}, \mathsf{false}, \mathsf{true}]\!] \\
&= (0, \tfrac{1}{4}]
\end{aligned}$$

♢

To use Theorem 8.22 to compute approximate preimages under $\mathsf{f_c}$ by computing images under its axial inverses, we must know whether each axis of $\mathsf{f_a}$ and $\mathsf{f_b}$ is increasing or decreasing. It helps to have an analogue of Lemma 8.6 (inverse direction).

Theorem 8.24. *Let* $\mathsf{f_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ *be uniformly surjective and strictly increasing or decreasing in each axis, with axial inverses* $\mathsf{f_a}$ *and* $\mathsf{f_b}$*. The following statements hold:*

1. $\mathsf{inc}_1\ \mathsf{f_a}$ *if and only if* $(\mathsf{inc}_1\ \mathsf{f_c})\ \mathsf{xor}\ (\mathsf{inc}_2\ \mathsf{f_c})$.
2. $\mathsf{inc}_2\ \mathsf{f_a}$ *if and only if* $\mathsf{inc}_1\ \mathsf{f_c}$.

Proof. For statement 1, let $\mathsf{c} \in \mathsf{C}$, $\mathsf{b}_1, \mathsf{b}_2 \in \mathsf{B}$, $\mathsf{a}_1 := \mathsf{f_a}\ \langle \mathsf{b}_1, \mathsf{c} \rangle$ and $\mathsf{a}_2 := \mathsf{f_a}\ \langle \mathsf{b}_2, \mathsf{c} \rangle$. Let $\mathsf{c}' := \mathsf{f_c}\ \langle \mathsf{a}_1, \mathsf{b}_2 \rangle$; note $\mathsf{c} = \mathsf{f_c}\ \langle \mathsf{a}_1, \mathsf{b}_1 \rangle = \mathsf{f_c}\ \langle \mathsf{a}_2, \mathsf{b}_2 \rangle$. Suppose $\mathsf{inc}_1\ \mathsf{f_c}$ and $\mathsf{inc}_2\ \mathsf{f_c}$; then $\mathsf{a}_1 > \mathsf{a}_2 \iff \mathsf{c} < \mathsf{c}'$ and $\mathsf{b}_1 < \mathsf{b}_2 \iff \mathsf{c} < \mathsf{c}'$, so $\mathsf{b}_1 < \mathsf{b}_2 \iff \mathsf{a}_1 > \mathsf{a}_2$. The remaining cases are similar.

For statement 2, fix $\mathsf{b} \in \mathsf{B}$ and apply Lemma 8.6. □

By Lemma 8.10, $\mathsf{inc}_1\ \mathsf{f_b} \iff \mathsf{inc}_2\ \mathsf{f_c}$, and $\mathsf{inc}_2\ \mathsf{f_b} \iff \mathsf{inc}_1\ \mathsf{f_a}$. We can therefore easily determine the uniform directions of $\mathsf{f_a}$'s and $\mathsf{f_b}$'s axes from the uniform directions of $\mathsf{f_c}$'s axes.

We are certain the preceeding definitions and theorems extend naturally to functions with any number of arguments, but have not needed to extend them yet.

8.2.3 Discontinuous Primitives

8.3 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 we must take care to round in the right directions and account for floating-point negative zero.

$$\begin{aligned}
&\overline{\mathsf{pos\text{-}recip}} : [0, +\infty] \to [0, +\infty] \\
&\overline{\mathsf{pos\text{-}recip}}\ \mathsf{a} \;=\; \mathsf{cond}\ \begin{array}{lll} 0 < \mathsf{a} < +\infty & \longrightarrow & 1/\mathsf{a} \\ \mathsf{a} = 0 & \longrightarrow & +\infty \\ \mathsf{a} = +\infty & \longrightarrow & 0 \end{array}
\end{aligned} \tag{8.14}$$

8.4 Partitioned Sampling

Suppose that we want to sample values in a probability space X, P by first sampling a set from a *partition* of X and then sampling from that set.[1] [XXX: this transition should mention why we want to do partition sampling instead of just supposing that we do]

First, we define

$$\begin{aligned}
&\mathsf{condition} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0,1] \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ \mathsf{X} \rightharpoonup [0,1] \\
&\mathsf{condition}\ \mathsf{P}\ \mathsf{A} \;:=\; \lambda \mathsf{A}' \in \mathsf{domain}\ \mathsf{P}.\, \mathsf{P}\ (\mathsf{A}' \cap \mathsf{A})\ /\ \mathsf{P}\ \mathsf{A}
\end{aligned} \tag{8.15}$$

to restrict a probability measure P to a measurable, positive-probability set A and renormalize it.

Definition 8.25 (partitioned sampling)**.** *Let* X, P *be an arbitrary probability space,* N *be an at-most-countable index set, and* $\mathsf{s} : \mathsf{N} \to \mathsf{Set}\ \mathsf{X}$ *be a partition of* X *into* $|\mathsf{N}|$ *measurable parts. The following procedure samples from* X*:*

1. *Choose* $\mathsf{n} \in \mathsf{N}$ *with probability* $\mathsf{P}\ (\mathsf{s}\ \mathsf{n})$.

[1] This is not *stratified* sampling, which samples a fixed number of times from each partition.

2. *Choose* $\mathsf{a} \in \mathsf{s}\ \mathsf{n}$ *according to* $\mathsf{condition}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n})$.

It is not hard to show that partitioned sampling chooses an $\mathsf{a} \in \mathsf{X}$ according to P.

Example 8.26 (partitioned sampling from a standard normal)**.** Let P be the standard normal distribution's probability measure. To sample according to P, let $\mathsf{N} := \{\mathsf{neg}, \mathsf{pos}\}$ and $\mathsf{s} = [\mathsf{neg} \mapsto (-\infty, 0], \mathsf{pos} \mapsto (0, +\infty)]$, and define $\mathsf{Q} : \mathsf{N} \to \mathsf{Set}\ \mathbb{R} \rightharpoonup [0, 1]$ by

$$\begin{aligned} \mathsf{Q}\ \mathsf{neg}\ \mathsf{A} &= \mathsf{P}\ ((-\infty, 0] \cap \mathsf{A})\ /\ \tfrac{1}{2} \\ \mathsf{Q}\ \mathsf{pos}\ \mathsf{A} &= \mathsf{P}\ ((0, +\infty) \cap \mathsf{A})\ /\ \tfrac{1}{2} \end{aligned} \tag{8.16}$$

Then

1. Choose $\mathsf{n} = \mathsf{neg}$ or $\mathsf{n} = \mathsf{pos}$, each with probability $\frac{1}{2}$.
2. Choose $\mathsf{a} \in \mathsf{s}\ \mathsf{n}$ according to $\mathsf{Q}\ \mathsf{n}$. ♢

Partitioned sampling has two weaknesses. First, it requires $\mathsf{P}\ (\mathsf{s}\ \mathsf{n})$ to be easy to compute for all $\mathsf{n} \in \mathsf{N}$. If this were true, we would not need to sample in the first place—i.e. it assumes a solution to the overall problem we are trying to solve. Second, it assumes sampling according to $\mathsf{condition}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n})$ 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 $\mathsf{s}\ \mathsf{n}$ for each $\mathsf{n} \in \mathsf{N}$. Doing so and returning weighted samples to adjust for the differences in distribution comprises ***partitioned importance sampling***.

First, we define

$$\begin{aligned} &\mathsf{subcond} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, 1] \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, 1] \\ &\mathsf{subcond}\ \mathsf{P}\ \mathsf{A} := \lambda \mathsf{A}' \in \mathsf{domain}\ \mathsf{P}.\ \mathsf{P}\ (\mathsf{A}' \cap \mathsf{A}) \end{aligned} \tag{8.17}$$

to restrict a probability measure P to a measurable set A *without* renormalizing it; i.e. it returns a subprobability measure.

Definition 8.27 (partitioned importance sampling)**.** *Suppose we have*

- *An arbitrary probability space* X, P.
- *An at-most-countable index set* N.
- *A probability mass function* $\mathsf{p} : \mathsf{N} \to [0, 1]$ *such that* $\mathsf{p}\ \mathsf{n} > 0$ *for all* $\mathsf{n} \in \mathsf{N}$.
- *A partition* $\mathsf{s} : \mathsf{N} \to \mathsf{Set}\ \mathsf{X}$ *of* X *into* $|\mathsf{N}|$ *measurable parts.*
- *Candidate probability measures* $\mathsf{Q} : \mathsf{N} \to \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, 1]$*, one for each partition.*

To sample from X *according to* P*,*

1. *Choose* $\mathsf{n} \in \mathsf{N}$ *with probability* $\mathsf{p}\ \mathsf{n}$.
2. *Choose* $\mathsf{a} \in \mathsf{X}$ *according to* $\mathsf{Q}\ \mathsf{n}$.
3. *Compute* $\mathsf{w} := \dfrac{1}{\mathsf{p}\ \mathsf{n}} \cdot \mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{a}$.
4. *Return the weighted sample* $\langle \mathsf{a}, \mathsf{w} \rangle$.

The function $\mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ (\mathsf{Q}\ \mathsf{n})$, with type $\mathsf{X} \to [0, +\infty)$, is a **Radon-Nikodým**[2] **derivative**. If P has density f, $\mathsf{Q}\ \mathsf{n}$ has density g with respect to the same base measure (e.g. same-dimensional length, area or volume), and $\mathsf{a} \in \mathsf{s}\ \mathsf{n}$ implies $\mathsf{g}\ \mathsf{a} > 0$, then[3]

$$\mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{a}\ =\ \mathsf{if}\ (\mathsf{a} \in \mathsf{s}\ \mathsf{n})\ (\mathsf{f}\ \mathsf{a}\ /\ \mathsf{g}\ \mathsf{a})\ 0 \tag{8.18}$$

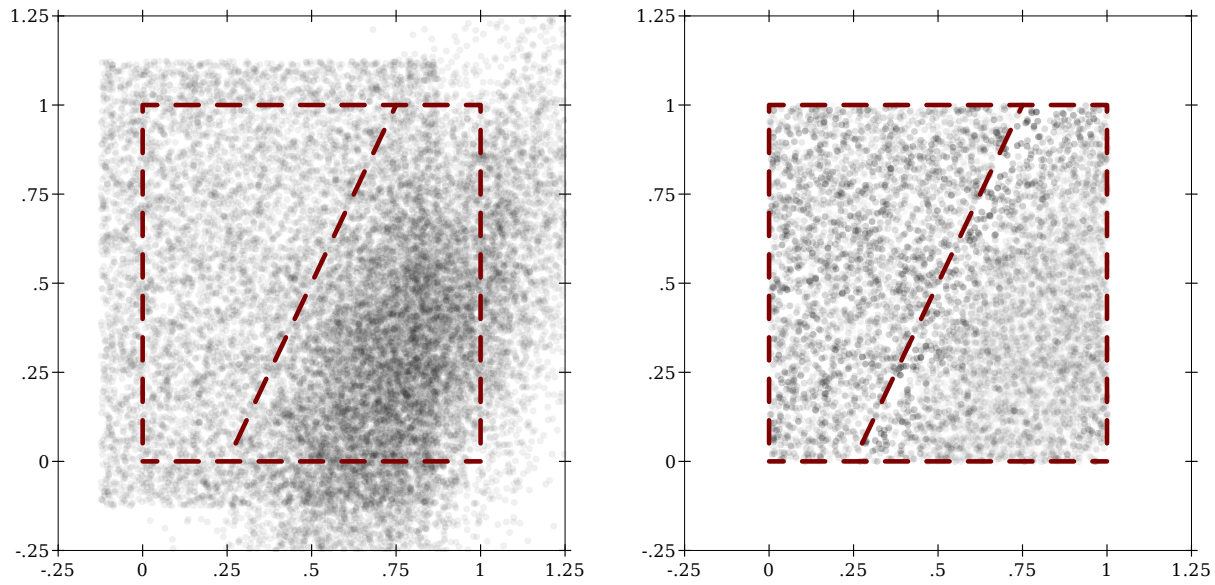
Chapter 10 has definitions and more details. We use diff^+ in a more general sense, but in this section, it is usually fine to think of its return values as quotients of densities.

An importance sampling algorithm is correct when all expected values computed using its weighted samples are the equal to the true expected values. The next theorem states that this is true of partitioned importance sampling under reasonable conditions, which are analogous to the support of $\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n})$ being no larger than that of $\mathsf{Q}\ \mathsf{n}$.

Theorem 8.28 (partitioned importance sampling correctness)**.** *Let* X*,* P*,* N*,* s*,* p*, and* Q *as in Definition 8.27 (partitioned importance sampling) such that* $\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}) \ll \mathsf{Q}\ \mathsf{n}$ *for all* $\mathsf{n} \in \mathsf{N}$*. Define* $\mathsf{P_N} : \mathsf{Set}\ \mathsf{N} \to [0, 1]$ *by extending* p *to a measure.*

[2]Pronounced "RADon neekohDIM," and named after Austrian mathematician Johann Radon and Polish mathematician Otto Nikodým.

[3]The equality in (8.18) holds $(\mathsf{Q}\ \mathsf{n})$-almost everywhere.



(a) Unweighted samples (b) Resampled by weight

Figure 8.4: Partitioned importance sampling used to sample uniformly in a partition of the unit square, using two overlapping candidate distributions.

If $\mathsf{g} : \mathsf{X} \to \mathbb{R}$ *is a* P*-integrable mapping, and*

$$\begin{aligned} &\mathsf{g}' : \mathsf{N} \times \mathsf{X} \to \mathbb{R} \\ &\mathsf{g}'\ \langle \mathsf{n}, \mathsf{a} \rangle\ :=\ \mathsf{g}\ \mathsf{a} \cdot \frac{1}{\mathsf{p}\ \mathsf{n}} \cdot \mathsf{diff}^{+}\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{a} \end{aligned} \tag{8.19}$$

then $\mathsf{int}\ \mathsf{g}'\ (\mathsf{P}_{\mathsf{N}} \times \mathsf{Q})\ (\mathsf{N} \times \mathsf{X})\ =\ \mathsf{int}\ \mathsf{g}\ \mathsf{P}\ \mathsf{X}$.

Proof. See Chapter 10. □

Partitioned importance sampling allows quite a lot of freedom: parts can be chosen with arbitrary nonzero probability, and each part can have its own candidate distribution, which may be defined on a superset of the part.

Example 8.29 (2D partitioned importance sampling)**.** Figure 8.4 shows the result of partitioned importance sampling in a partition of the unit square. In this instance,

- $\mathsf{X} := [0, 1] \times [0, 1]$ and P is the uniform measure on X (i.e. area).
- $\mathsf{N} := \{\mathsf{left}, \mathsf{right}\}$.
- $\mathsf{p} := [\mathsf{left} \mapsto 0.4, \mathsf{right} \mapsto 0.6]$.

- $\mathsf{s\ left} = \{\langle \mathsf{x}, \mathsf{y}\rangle \in \mathsf{X} \mid \mathsf{y} > 2 \cdot \mathsf{x} - \frac{1}{2}\}$, similarly for $\mathsf{s\ right}$.
- $\mathsf{Q\ left}$ is the uniform measure on a superset of $\mathsf{s\ left}$, and $\mathsf{Q\ right}$ is a multivariate Gaussian measure centered at $\langle 0.8, 0.3\rangle$.

The implementation does not actually construct most of these objects. It constructs

- A density function $\mathsf{f} : \mathbb{R} \times \mathbb{R} \Rightarrow [0, +\infty)$ to represent P.
- A family of predicates $\mathsf{s?} : \mathsf{N} \Rightarrow \mathsf{X} \Rightarrow \mathsf{Bool}$ to decide $\mathsf{a} \in \mathsf{s\ n}$.
- Candidate densities $\mathsf{g} : \mathsf{N} \Rightarrow \mathsf{X} \Rightarrow [0, +\infty)$ to represent Q.

It computes weights using $\mathsf{diff}^+\ (\mathsf{subcond\ P\ (s\ n)})\ (\mathsf{Q\ n})\ \mathsf{a} = \mathsf{f\ a} \,/\, \mathsf{g\ n\ a}$ when $\mathsf{s?\ n\ a} = \mathsf{true}$. It directly represents only N and p, but we will find even this to be infeasible shortly. ♢

Two properties make the preceeding example simple. First, the partition has finitely many parts. Second, the measures $\mathsf{subcond\ P\ (s\ n)}$ and $\mathsf{Q\ n}$ have densities with respect to the same base measure, which ensures $\mathsf{diff}^+\ (\mathsf{subcond\ P\ (s\ n)})\ (\mathsf{Q\ n})$ exists and is easy to compute.

When sampling in the domain of programs, neither property holds in general.

8.4.1 Partitioning Probabilistic Program Domains

For the random source part $\mathsf{R} := \mathsf{J} \to [0, 1]$ of probabilistic language domains, which consists of infinite binary trees of reals, it is not clear that Theorem 8.28 is applicable. The main problem is that infinite-dimensional Radon-Nikodým derivatives do not generally exist.

Fortunately, they *can* exist if the two measures differ in only finitely many axes. More precisely, let $\mathsf{P}_1 : \mathsf{Set\ R} \to [0, 1]$ and $\mathsf{P}_2 : \mathsf{Set\ R} \to [0, 1]$ be probability measures, and $\mathsf{J}' \subseteq \mathsf{J}$ be a finite set of tree indexes. Suppose P_1 can be factored into a distribution P'_1 over $\mathsf{J}' \to [0, 1]$ and a distribution over $(\mathsf{J} \backslash \mathsf{J}') \to [0, 1]$, and P'_2 can be similarly factored into P'_2 and *the same* distribution over $(\mathsf{J} \backslash \mathsf{J}') \to [0, 1]$. Then, under reasonable conditions (which are analogous to the support of P'_1 being no larger than that of P'_2), $\mathsf{diff}^+\ \mathsf{P}_1\ \mathsf{P}_2$ exists and can be computed using $\mathsf{diff}^+\ \mathsf{P}'_1\ \mathsf{P}'_2$. Chapter 10 contains a formal statement of this fact and a proof.

To ensure $\mathsf{subcond\ P\ (s\ n)}$ and $\mathsf{Q\ n}$ differ in only finitely many axes, we partition R according to branch traces. Each branch trace corresponds with a program that reads any

$\mathsf{r} \in \mathsf{R}$ at only finitely many indexes $\mathsf{J}' \subseteq \mathsf{J}$. [XXX: connect idea better]

In the remainder of this subsection, assume a fixed program p. Let $\mathsf{f} := [\![p]\!]^{\Downarrow}_{\perp^*} : \langle\langle \mathsf{R}, \mathsf{T}\rangle, \langle\rangle\rangle \rightsquigarrow_{\perp^*} \mathsf{Y}$ be its interpretation as a bottom* arrow computation, with maximal domain A^*. Define $\mathsf{T}^* := \mathsf{image}\ (\mathsf{fst} \ggg \mathsf{snd})\ \mathsf{A}^*$ as its ***maximal branch trace set*** and $\mathsf{R}^* := \mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ \mathsf{A}^*$ as its ***maximal random source set***.

We need a notion of the random sources that agree with a given branch trace $\mathsf{t} \in \mathsf{T}$; i.e. those $\mathsf{r} \in \mathsf{R}$ for which $\mathsf{f}\ \langle\langle \mathsf{r}, \mathsf{t}\rangle, \langle\rangle\rangle \neq \perp$.

Definition 8.30 (induced random sources)**.** *Let* $\mathsf{t} \in \mathsf{T}$ *be a branch trace. The* ***random sources induced by*** t *are a subset of* R *defined by* $\mathsf{R}' := \{\mathsf{r} \in \mathsf{R} \mid \langle\langle \mathsf{r}, \mathsf{t}\rangle, \langle\rangle\rangle \in \mathsf{A}^*\}$.

[XXX: graph of induced partition from program $\mathsf{if}\ (\mathsf{random} < \mathsf{random})\ \mathsf{true}\ \mathsf{false}$?]

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 8.31. For the program

$$\mathsf{if}\ (\mathsf{random} < \mathsf{p})\ 0\ \mathsf{random} \tag{8.20}$$

there are at least two branch traces in the program's maximal domain: $\mathsf{t}_0 := [\mathsf{j}_0 \mapsto \mathsf{true}, * \mapsto \perp]$ and $\mathsf{t}_1 := [\mathsf{j}_0 \mapsto \mathsf{false}, * \mapsto \perp]$. There is also $[\mathsf{j}_0 \mapsto \mathsf{false}, \mathsf{left}\ \mathsf{j}_0 \mapsto \mathsf{true}, * \mapsto \perp]$, because it agrees with every execution that $[\mathsf{j}_0 \mapsto \mathsf{false}, * \mapsto \perp]$ 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?] ♢

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.

First, we need to be sure that such equivalence classes can induce a partition.

Theorem 8.32. *Let* $\mathsf{t}_1, \mathsf{t}_2 \in \mathsf{T}^*$. *If* t_1 *induces* R'_1 *and* t_2 *induces* R'_2, *then* $\mathsf{R}'_1 = \mathsf{R}'_2$ *or* $\mathsf{R}'_1 \cap \mathsf{R}'_2 = \varnothing$.

Proof. XXX: do this □

To identify the "smallest" trace in each class, we must define an ordering over them. One fairly natural way is to say a branch trace is smaller than another when it describes fewer branch decisions; i.e. its tree has fewer non-$\bot$ elements. Two branch traces that differ by returning respectively true and false for the same j may represent different execution paths, so they must be incomparable.

Definition 8.33 (brach trace partial order)**.** $\mathsf{t}_1 \leq \mathsf{t}_2$ *when for all* $\mathsf{j} \in \mathsf{J}$, *if* $\mathsf{t}_1\ \mathsf{j} \neq \mathsf{t}_2\ \mathsf{j}$, *then* $\mathsf{t}_1\ \mathsf{j} = \bot$.

If T^* is partitioned into equivalence classes of traces that induce the same random sources, each part in the partition contains a least member with respect to $(\leq)$.

Theorem 8.34. *Let* $\mathsf{t} \in \mathsf{T}^*$ *induce* R', *and let* T' *be the largest subset of* T^* *that induces* R'. T' *has a least member* t_*.

Proof. Define $\mathsf{t}_* \in \mathsf{T}$ by

$$\begin{array}{rcll} \mathsf{t}_* & := & \mathsf{cond} & \forall\, \mathsf{t}' \in \mathsf{T}'.\, \mathsf{t}'\ \mathsf{j} = \mathsf{true} & \longrightarrow & \mathsf{true} \\ & & & \forall\, \mathsf{t}' \in \mathsf{T}'.\, \mathsf{t}'\ \mathsf{j} = \mathsf{false} & \longrightarrow & \mathsf{false} \\ & & & \mathsf{else} & \longrightarrow & \bot \end{array} \tag{8.21}$$

Thus, $\mathsf{t}_* \leq \mathsf{t}'$ for all $\mathsf{t}' \in \mathsf{T}'$, or t_* is a lower bound for T'.

Let $\mathsf{r} \in \mathsf{R}'$. By construction, every conditional subcomputation at index $\mathsf{j} \in \mathsf{J}$ agrees with $\mathsf{t}_*\ \mathsf{j}$. Therefore $\mathsf{f}\ \langle\langle \mathsf{r}, \mathsf{t}_* \rangle, \langle\rangle\rangle \neq \bot$, so $\langle\langle \mathsf{r}, \mathsf{t}_* \rangle, \langle\rangle\rangle \in \mathsf{A}^*$, and thus $\mathsf{t}_* \in \mathsf{T}'$. □

An easy consequence is $\mathsf{T}' = \{\mathsf{t}' \in \mathsf{T}^* \mid \mathsf{t}_* \leq \mathsf{t}'\}$. [XXX: is it easy?] Thus, each least member represents a set of larger branch traces that induce the same set of random sources.

We can get our sought-after index set by defining the set of smallest branch traces.

Definition 8.35 (minimal branch traces)**.** *The set of* ***minimal branch traces*** T_* *is the set of minimal elements in* T^**, or*

$$\mathsf{T}_* \;:=\; \{\mathsf{t}_1 \in \mathsf{T}^* \mid \forall\, \mathsf{t}_2 \in \mathsf{T}^*.\, \mathsf{t}_2 \leq \mathsf{t}_1 \implies \mathsf{t}_2 = \mathsf{t}_1\} \tag{8.22}$$

Theorem 8.36. T_* *induces a partition of* R^**.*

Proof. XXX: todo □

We can infer from Theorem 8.36 that a program's minimal branch trace set T_* contains only the actual branches taken when running the program on every $\mathsf{r} \in \mathsf{R}^*$. 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 $\mathsf{r} \in \mathsf{R}$ uniformly, and run the program on r while recording each branch decision.

But this sampling scheme has problems similar to those of partitioned sampling (Definition 8.25). First, it assumes the probabilities of branch traces, which are the probabilities of the R^* subsets they induce, are easy to compute. Second, we are interested in sampling from an *arbitrarily low-probability subset* of R^*, 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 R^* requires sampling in a small subset of T_*.
2. Sampling in a small subset of T_* requires sampling in a small subset of R^*.

Fortunately, if we allow ourselves subsets of a slightly larger set than T_*, and allow ourselves to sample within overapproximating *covers* of R^* subsets, we can use approximate preimage computation to sample from T_* and R^* subsets simultaneously.

8.4.2 Approximate Partitions of Probabilistic Program Domains

We first define the set of branch traces that is slightly larger than T_*.

The idea is to define a set of *feasible* branch traces T_+ that is derived only from the

shape of a program, not its actual executions. We must then ensure that every additional branch trace (i.e. every $\mathsf{t} \in \mathsf{T}_+ \backslash \mathsf{T}_*$) induces $\varnothing$, so that T_+ induces a partition. After that, we define an algorithm for sampling from T_+, which does not require running a probabilistic program on any $\mathsf{r} \in \mathsf{R}$, and prove the algorithm correct. We can then extend this algorithm to use preimage computation to sample in arbitrarily good approximations of small subsets of T_* and R^*.

Defining T_+ in terms of a program's abstract branching shape requires an additional nonstandard interpretation. Figure 8.5a defines the ***branch index*** arrow. Its type is $\mathsf{J} \Rightarrow \mathsf{Idxs}$, which does not refer to a domain or codomain type of program values because its computations do not receive or compute program values. Instead, they build lazy trees of possible branching decisions, ignoring the actual values of if conditions. For example, lifted, pure functions are interpreted as $\lambda \mathsf{j}.\, \langle\rangle$, which takes the function's computation index and returns no decisions. Composition and pairing of subcomputations k_1 and k_2 both return $\langle \mathsf{k}_1\ (\mathsf{left}\ \mathsf{j}), \mathsf{k}_2\ (\mathsf{right}\ \mathsf{j}) \rangle$: a node with two children that contain the feasible branch decisions in their subcomputations.

Only $\mathsf{ifte}_{\mathsf{idxs}^*}$ does more than simple structural recursion: it returns $\mathsf{if\text{-}idxs}\ \mathsf{j}\ \mathsf{idxs}_2\ \mathsf{idxs}_3$ 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 8.5b defines the function traces, which transforms these lazy trees of type Idxs into sets of branch traces. To ensure trace always terminates, it is defined using a depth-limited helper function, which it calls with every depth $\mathsf{n} \in \mathbb{N}$, and collects the results in a countable union. It is thus unimplementable, but we will use it only to precisely define the range of a random value.

We define a program's feasible branch traces using the standard first-order semantic function $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a}}$ with arrow $\mathsf{a} = \mathsf{idxs}^*$.

Definition 8.37 (feasible branch traces)**.** *A program p's* ***feasible branch traces*** *are those*

$$
\begin{aligned}
&\mathsf{Idxs} ::= \langle\rangle \mid \langle \mathsf{Idxs}, \mathsf{Idxs}\rangle \\
&\qquad \mid \mathsf{if\text{-}idxs\ J\ (1 \Rightarrow Idxs)\ (1 \Rightarrow Idxs)} \\
&\\
&\mathsf{arr}_{\mathsf{idxs}^*} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \\
&\mathsf{arr}_{\mathsf{idxs}^*}\ \mathsf{f\ j} := \langle\rangle \\
&\\
&(\ggg_{\mathsf{idxs}^*}) : (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \\
&(\mathsf{k}_1 \ggg_{\mathsf{idxs}^*} \mathsf{k}_2)\ \mathsf{j} := \langle \mathsf{k}_1\ (\mathsf{left\ j}), \mathsf{k}_2\ (\mathsf{right\ j})\rangle \\
&\\
&(\&\&\&_{\mathsf{idxs}^*}) : (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \\
&(\mathsf{k}_1 \ggg_{\mathsf{idxs}^*} \mathsf{k}_2)\ \mathsf{j} := \langle \mathsf{k}_1\ (\mathsf{left\ j}), \mathsf{k}_2\ (\mathsf{right\ j})\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{ifte}_{\mathsf{idxs}^*} : (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \\
&\mathsf{ifte}_{\mathsf{idxs}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} := \mathsf{let}\ \ \mathsf{idxs}_2 := \lambda 0.\, \mathsf{k}_2\ (\mathsf{left\ (right\ j)}) \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{idxs}_3 := \lambda 0.\, \mathsf{k}_3\ (\mathsf{right\ (right\ j)}) \\
&\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \langle \mathsf{k}_1\ \mathsf{j}, \mathsf{if\text{-}idxs\ j\ idxs}_2\ \mathsf{idxs}_3\rangle \\
&\\
&\mathsf{lazy}_{\mathsf{idxs}^*} : (1 \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs})) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \\
&\mathsf{lazy}_{\mathsf{idxs}^*}\ \mathsf{k\ j} := \mathsf{k}\ 0\ \mathsf{j} \\
&\\
&\mathsf{random}_{\mathsf{idxs}^*} : \mathsf{J} \Rightarrow \mathsf{Idxs} \\
&\mathsf{random}_{\mathsf{idxs}^*}\ \mathsf{j} := \langle\rangle
\end{aligned}
$$

(a) Branch index arrow. Computations return a lazy tree of type Idxs, of feasible branch decisions, ignoring the actual values of if conditions. The arrow is directly implementable in any λ-calculus.

$$
\begin{aligned}
&\mathsf{traces} : \mathsf{Idxs} \Rightarrow \mathsf{Set\ T} \\
&\mathsf{traces\ idxs} := \textstyle\bigcup_{n \in \mathbb{N}} \mathsf{traces}^*\ \mathsf{n\ idxs}\ [* \mapsto \bot] \\
&\\
&\mathsf{traces}^* : \mathbb{N} \Rightarrow \mathsf{Idxs} \Rightarrow \mathsf{T} \Rightarrow \mathsf{Set\ T} \\
&\mathsf{traces}^*\ \mathsf{n}\ \langle\rangle\ \mathsf{t} := \{\mathsf{t}\} \\
&\mathsf{traces}^*\ \mathsf{n}\ \langle \mathsf{idxs}_1, \mathsf{idxs}_2\rangle\ \mathsf{t} := \mathsf{let}\ \ \mathsf{T}' := \mathsf{traces}^*\ \mathsf{n\ idxs}_1\ \mathsf{t} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \textstyle\bigcup_{\mathsf{t}' \in \mathsf{T}'} \mathsf{traces}^*\ \mathsf{n\ idxs}_2\ \mathsf{t}' \\
&\mathsf{traces}^*\ 0\ (\mathsf{if\text{-}idxs\ j\ idxs}_2\ \mathsf{idxs}_3)\ \mathsf{t} := \{\mathsf{t}\} \\
&\mathsf{traces}^*\ \mathsf{n}\ (\mathsf{if\text{-}idxs\ j\ idxs}_2\ \mathsf{idxs}_3)\ \mathsf{t} := \mathsf{traces}^*\ (\mathsf{n}-1)\ (\mathsf{idxs}_2\ 0)\ \mathsf{t}[\mathsf{j} \mapsto \mathsf{true}] \cup \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{traces}^*\ (\mathsf{n}-1)\ (\mathsf{idxs}_3\ 0)\ \mathsf{t}[\mathsf{j} \mapsto \mathsf{false}] \cup \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \{\mathsf{t}\}
\end{aligned}
\tag{8.23}
$$

(b) The λ_{ZFC} function traces turns a lazy tree of feasible branch decisions into a set of feasible branch traces.

$$
\begin{aligned}
&\mathsf{sample\text{-}trace} : \mathsf{Idxs} \to \langle \mathbb{R}, \mathsf{T}\rangle \\
&\mathsf{sample\text{-}trace\ idxs} := \mathsf{sample\text{-}trace}^*\ \mathsf{idxs}\ \langle 1, [* \mapsto \bot]\rangle \\
&\\
&\mathsf{sample\text{-}trace}^* : \mathsf{Idxs} \Rightarrow \langle \mathbb{R}, \mathsf{T}\rangle \Rightarrow \langle \mathbb{R}, \mathsf{T}\rangle \\
&\mathsf{sample\text{-}trace}^*\ \langle\rangle\ \mathsf{pt} := \mathsf{pt} \\
&\mathsf{sample\text{-}trace}^*\ \langle \mathsf{idxs}_1, \mathsf{idxs}_2\rangle\ \mathsf{pt} := \mathsf{let}\ \ \mathsf{pt}' := \mathsf{sample\text{-}trace}^*\ \mathsf{idxs}_1\ \mathsf{pt} \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \mathsf{sample\text{-}trace}^*\ \mathsf{idxs}_2\ \mathsf{pt}' \\
&\mathsf{sample\text{-}trace}^*\ (\mathsf{if\text{-}idxs\ j\ idxs}_2\ \mathsf{idxs}_3)\ \langle \mathsf{p_t}, \mathsf{t}\rangle := \mathsf{let}\ \ \langle \mathsf{p_b}, \mathsf{b}\rangle := \mathsf{sample\text{-}branch}\ \{\mathsf{true}, \mathsf{false}, \bot\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathsf{pt}' := \langle \mathsf{p_t} \cdot \mathsf{p_b}, \mathsf{t}[\mathsf{j} \mapsto \mathsf{b}]\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \mathsf{case\ b} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{true} \longrightarrow \mathsf{sample\text{-}trace}^*\ (\mathsf{idxs}_2\ 0)\ \mathsf{pt}' \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{false} \longrightarrow \mathsf{sample\text{-}trace}^*\ (\mathsf{idxs}_3\ 0)\ \mathsf{pt}' \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \bot \longrightarrow \mathsf{pt}'
\end{aligned}
\tag{8.24}
$$

(c) The stochastic function sample-trace samples t from the set returned by traces, and returns t and its probability. It is directly implementable in any λ-calculus with probabilistic choice.

Figure 8.5: Branch index collecting semantics.

in the set $\mathsf{T}_+ := \mathsf{traces}\ (\llbracket p \rrbracket^{\Downarrow}_{\mathsf{idxs}^*}\ \mathsf{j}_0)$.

If we are to sample $\mathsf{t} \in \mathsf{T}_+$ and then sample within the R' induced by t, we must ensure that T_+ includes T_* and induces a partition.

Theorem 8.38. *Let* $\mathsf{t} \in \mathsf{T}$ *induce* R'. $\mathsf{R}' \neq \varnothing$ *if and only if* $\mathsf{t} \in \mathsf{T}^*$. *[XXX: needed?]*

Proof. $\mathsf{R}' \neq \varnothing$ if and only if there is an $\mathsf{r} \in \mathsf{R}'$ such that $\langle\langle \mathsf{r}, \mathsf{t}\rangle, \langle\rangle\rangle \in \mathsf{A}^*$. □

Theorem 8.39. *For all* $\mathsf{t} \in \mathsf{T}_+$, *either* $\mathsf{t} \in \mathsf{T}_*$ *or* t *induces* $\varnothing$.

Proof. XXX: todo □

Corollary 8.40. T_+ *induces a partition of* R^*.

XXX: now we define a stochastic procedure for sampling in T_+

XXX: apologize for being less precise (e.g. stochastic procedures look like λ_{ZFC} but aren't; we assume every operation is lifted to receive a random source as input, etc.)

XXX: suppose we have a stochastic procedure $\mathsf{sample\text{-}branch} : \mathsf{Set}\ \mathsf{Bool}_\perp \Rightarrow \langle \mathbb{R}, \mathsf{Bool}_\perp \rangle$, where $\mathsf{sample\text{-}branch}\ \mathsf{B}$ returns any member of B with some nonzero, constant probability

Figure 8.5c defines $\mathsf{sample\text{-}trace}$, a stochastic procedure defined in terms of $\mathsf{sample\text{-}branch}$. Given an $\mathsf{idxs} : \mathsf{Idxs}$, it returns a member of $\mathsf{traces}\ \mathsf{idxs}$ and its probability. In other words, for a program p, the probability that $\langle \mathsf{p_t}, \mathsf{t} \rangle = \mathsf{sample\text{-}trace}\ (\llbracket p \rrbracket^{\Downarrow}_{\mathsf{idxs}^*}\ \mathsf{j}_0)$ is $\mathsf{p_t}$, and $\mathsf{t} \in \mathsf{T}_+$. Further, every $\mathsf{t} \in \mathsf{T}_+$ has a constant, nonzero probability of being chosen.

Of course, we must prove these facts.

XXX: move the following soundness and completeness proofs to Chapter 10, or define T_+ in terms of $\mathsf{sample\text{-}trace}$ to make the proofs unnecessary; whether the latter will work (or whether it will work better) will probably become clear when I do the missing proof above

Theorem 8.41 ($\mathsf{sample\text{-}trace}^*$ soundness)**.** *For all* $\mathsf{t} \in \mathsf{T}$ *and* $\mathsf{idxs} : \mathsf{Idxs}$, *there exists an* $\mathsf{n} \in \mathbb{N}$ *such that* $\mathsf{snd}\ (\mathsf{sample\text{-}trace}^*\ \mathsf{idxs}\ \langle \mathsf{p_t}, \mathsf{t} \rangle) \in \mathsf{traces}^*\ \mathsf{n}\ \mathsf{idxs}\ \mathsf{t}$.

Proof. By structural induction on Idxs.

Case $\mathsf{idxs} = \langle\rangle$. For all $\mathsf{n} \in \mathbb{N}$, these two statements are equivalent by substituting definitions of $\mathsf{sample\text{-}trace}^*$ and traces^*:

$$\begin{gathered}\mathsf{snd}\ (\mathsf{sample\text{-}trace}^*\ \langle\rangle\ \langle \mathsf{p_t}, \mathsf{t}\rangle) \in \mathsf{traces}^*\ \mathsf{n}\ \langle\rangle\ \mathsf{t} \\ \mathsf{t} \in \{\mathsf{t}\}\end{gathered} \tag{8.25}$$

Case $\mathsf{idxs} = \langle \mathsf{idxs}_1, \mathsf{idxs}_2\rangle$. Let

$$\begin{aligned}\langle \mathsf{p}'_\mathsf{t}, \mathsf{t}'\rangle &:= \mathsf{sample\text{-}trace}^*\ \mathsf{idxs}_1\ \langle \mathsf{p_t}, \mathsf{t}\rangle \\ \langle \mathsf{p}''_\mathsf{t}, \mathsf{t}''\rangle &:= \mathsf{sample\text{-}trace}^*\ \mathsf{idxs}_2\ \langle \mathsf{p}'_\mathsf{t}, \mathsf{t}'\rangle\end{aligned} \tag{8.26}$$

as in the definition of $\mathsf{sample\text{-}trace}^*$. By hypothesis, there exists an $\mathsf{n}' \in \mathbb{N}$ such that $\mathsf{t}' \in \mathsf{traces}^*\ \mathsf{n}'\ \mathsf{idxs}_1\ \mathsf{t}$. Again by hypothesis, there exists an $\mathsf{n}'' \in \mathbb{N}$ such that $\mathsf{t}'' \in \mathsf{traces}^*\ \mathsf{n}''\ \mathsf{idxs}_2\ \mathsf{t}'$. Let $\mathsf{n} := \mathsf{max}\ \mathsf{n}'\ \mathsf{n}''$.

Case $\mathsf{idxs} = \mathsf{if\text{-}idxs}\ \mathsf{j}\ \mathsf{idxs}_2\ \mathsf{idxs}_3$. Let

$$\begin{aligned}\langle \mathsf{p_b}, \mathsf{b}\rangle &:= \mathsf{sample\text{-}branch}\ \{\mathsf{true}, \mathsf{false}, \bot\} \\ \mathsf{pt}' &:= \langle \mathsf{p_t} \cdot \mathsf{p_b}, \mathsf{t}[\mathsf{j} \mapsto \mathsf{b}]\rangle\end{aligned} \tag{8.27}$$

as in the definition of $\mathsf{sample\text{-}trace}^*$, and let $\langle \mathsf{p}'_\mathsf{t}, \mathsf{t}'\rangle = \mathsf{pt}'$. If $\mathsf{b} = \mathsf{true}$, by hypothesis, there exists an $\mathsf{n}' \in \mathbb{N}$ such that $\mathsf{snd}\ (\mathsf{sample\text{-}trace}^*\ (\mathsf{idxs}_2\ 0)\ \mathsf{pt}') \in \mathsf{traces}^*\ \mathsf{n}'\ (\mathsf{idxs}_2\ 0)\ \mathsf{t}'$; similarly for $\mathsf{b} = \mathsf{false}$. Let $\mathsf{n} := \mathsf{n}' + 1$, so $\mathsf{n}' = \mathsf{n} - 1$. If $\mathsf{b} = \bot$, then $\mathsf{t}' = \mathsf{t}$, and $\mathsf{t} \in \{\mathsf{t}\}$ for all $\mathsf{n} \in \mathbb{N}$. □

Corollary 8.42 ($\mathsf{sample\text{-}trace}$ soundness)**.** *For all* $\mathsf{idxs} : \mathsf{Idxs}$, $\mathsf{snd}\ (\mathsf{sample\text{-}trace}\ \mathsf{idxs}) \in \mathsf{traces}\ \mathsf{idxs}$.

Theorem 8.43 ($\mathsf{sample\text{-}trace}^*$ completeness)**.** *For all* $\mathsf{idxs} : \mathsf{Idxs}$, $\mathsf{t} \in \mathsf{T}$ *and* $\mathsf{n} \in \mathbb{N}$, $\mathsf{t}' \in \mathsf{traces}^*\ \mathsf{n}\ \mathsf{idxs}\ \mathsf{t}$ *implies* $\mathsf{Pr}[\mathsf{snd}\ (\mathsf{sample\text{-}trace}^*\ \mathsf{idxs}\ \langle \mathsf{p_t}, \mathsf{t}\rangle) = \mathsf{t}'] > 0$ *(or just* $\mathsf{Pr}[\mathsf{t}'] > 0$*).*

Proof. By structural induction on Idxs.

Case $\mathsf{idxs} = \langle\rangle$. By definition of traces^*, $\mathsf{t}' \in \mathsf{traces}^*\ \mathsf{n}\ \langle\rangle\ \mathsf{t}$ if and only if $\mathsf{t}' \in \{\mathsf{t}\}$, or $\mathsf{t}' = \mathsf{t}$. By definition of $\mathsf{sample\text{-}trace}^*$, $\mathsf{Pr}[\mathsf{snd}\ (\mathsf{sample\text{-}trace}^*\ \mathsf{idxs}\ \langle \mathsf{p_t}, \mathsf{t}\rangle) = \mathsf{t}] = 1$.

Case $\mathsf{idxs} = \langle \mathsf{idxs}_1, \mathsf{idxs}_2 \rangle$. Let

$$\begin{aligned} \mathsf{T}' &:= \mathsf{traces}^*\ \mathsf{n}\ \mathsf{idxs}_1\ \mathsf{t} \\ \mathsf{T}'' &:= \textstyle\bigcup_{\mathsf{t}' \in \mathsf{T}'} \mathsf{traces}^*\ \mathsf{n}\ \mathsf{idxs}_2\ \mathsf{t}' \end{aligned} \tag{8.28}$$

as in the definition of traces^*. Let

$$\begin{aligned} \langle \mathsf{p}'_\mathsf{t}, \mathsf{t}' \rangle &:= \mathsf{sample\text{-}trace}^*\ \mathsf{idxs}_1\ \langle \mathsf{p}_\mathsf{t}, \mathsf{t} \rangle \\ \langle \mathsf{p}''_\mathsf{t}, \mathsf{t}'' \rangle &:= \mathsf{sample\text{-}trace}^*\ \mathsf{idxs}_2\ \langle \mathsf{p}'_\mathsf{t}, \mathsf{t}' \rangle \end{aligned} \tag{8.29}$$

as in the definition of $\mathsf{sample\text{-}trace}^*$. Suppose $\mathsf{t}'' \in \mathsf{T}''$. Then there exists a $\mathsf{t}' \in \mathsf{T}'$ such that $\mathsf{t}'' \in \mathsf{traces}^*\ \mathsf{n}\ \mathsf{idxs}_2\ \mathsf{t}'$. By hypothesis, $\mathsf{Pr}[\mathsf{t}'] > 0$, therefore by hypothesis, $\mathsf{Pr}[\mathsf{t}''] > 0$.

Case $\mathsf{idxs} = \mathsf{if\text{-}idxs}\ \mathsf{j}\ \mathsf{idxs}_2\ \mathsf{idxs}_3$. Let

$$\begin{aligned} \langle \mathsf{p}_\mathsf{b}, \mathsf{b} \rangle &:= \mathsf{sample\text{-}branch}\ \{\mathsf{true}, \mathsf{false}, \bot\} \\ \mathsf{pt}' &:= \langle \mathsf{p}_\mathsf{t} \cdot \mathsf{p}_\mathsf{b}, \mathsf{t}[\mathsf{j} \mapsto \mathsf{b}] \rangle \end{aligned} \tag{8.30}$$

as in the definition of $\mathsf{sample\text{-}trace}^*$, and let $\langle \mathsf{p}'_\mathsf{t}, \mathsf{t}' \rangle = \mathsf{pt}'$.

If $\mathsf{n} = 0$, then let

$$\mathsf{T}' := \mathsf{traces}^*\ 0\ \mathsf{idxs}\ \mathsf{t} = \{\mathsf{t}\} \tag{8.31}$$

as in the definition of traces^*. Suppose $\mathsf{t}' \in \mathsf{T}'$, equiv. $\mathsf{t}' = \mathsf{t}$, so $\mathsf{Pr}[\mathsf{t}'] > 0$ iff $\mathsf{Pr}[\mathsf{t}] > 0$. Because $\mathsf{Pr}[\mathsf{b} = \bot] > 0$ and $\mathsf{t}[\mathsf{j} \mapsto \bot] = \mathsf{t}$, $\mathsf{Pr}[\mathsf{t}] > 0$.

If $\mathsf{n} > 0$, then expand $\mathsf{traces}^*\ \mathsf{n}\ \mathsf{idxs}\ \mathsf{t}$ to

$$\begin{aligned} \mathsf{T}'_\mathsf{true} &:= \mathsf{traces}^*\ (\mathsf{n} - 1)\ (\mathsf{idxs}_2\ 0)\ \mathsf{t}[\mathsf{j} \mapsto \mathsf{true}] \\ \mathsf{T}'_\mathsf{false} &:= \mathsf{traces}^*\ (\mathsf{n} - 1)\ (\mathsf{idxs}_3\ 0)\ \mathsf{t}[\mathsf{j} \mapsto \mathsf{false}] \\ \mathsf{T}'_\bot &:= \{\mathsf{t}\} \\ \mathsf{T}' &:= \mathsf{T}'_\mathsf{true} \cup \mathsf{T}'_\mathsf{false} \cup \mathsf{T}'_\bot \end{aligned} \tag{8.32}$$

as in the definition of traces^*. Suppose $\mathsf{t}' \in \mathsf{T}'$. If $\mathsf{t}' \in \mathsf{T}'_\mathsf{true}$, then because $\mathsf{Pr}[\mathsf{b} = \mathsf{true}] > 0$, by hypothesis, $\mathsf{Pr}[\mathsf{t}'] > 0$; similarly for $\mathsf{t}' \in \mathsf{T}'_\mathsf{false}$ and $\mathsf{t}' \in \mathsf{T}'_\bot$. □

Corollary 8.44 (sample-trace completeness)**.** *For all* $\mathsf{idxs} : \mathsf{Indx}$, $\mathsf{t} \in \mathsf{traces\ idxs}$ *implies* $\mathsf{Pr[snd\ (sample\text{-}trace\ idxs) = t]} > 0$.

Theorem 8.45 (sample-trace* correctness)**.** *Let* $\mathsf{idxs} : \mathsf{Idxs}$, $\mathsf{t} \in \mathsf{T}_+$, $\mathsf{p_t} \in [0, 1]$, *and* $\langle \mathsf{p'_t}, \mathsf{t'} \rangle := \mathsf{sample\text{-}trace^*\ idxs}\ \langle \mathsf{p_t}, \mathsf{t} \rangle$. *If* $\mathsf{Pr[t]} = \mathsf{p_t}$, *then* $\mathsf{Pr[t']} = \mathsf{p'_t}$.

Proof. By structural induction on Idxs.

Case $\mathsf{idxs} = \langle\rangle$. By definition of sample-trace*, $\langle \mathsf{p'_t}, \mathsf{t'} \rangle = \langle \mathsf{p_t}, \mathsf{t} \rangle$. Clearly $\mathsf{Pr[t]} = \mathsf{p_t}$ implies $\mathsf{Pr[t']} = \mathsf{p'_t}$.

Case $\mathsf{idxs} = \langle \mathsf{idxs}_1, \mathsf{idxs}_2 \rangle$. Let

$$\begin{aligned} \langle \mathsf{p'_t}, \mathsf{t'} \rangle &:= \mathsf{sample\text{-}trace^*\ idxs}_1\ \langle \mathsf{p_t}, \mathsf{t} \rangle \\ \langle \mathsf{p''_t}, \mathsf{t''} \rangle &:= \mathsf{sample\text{-}trace^*\ idxs}_2\ \langle \mathsf{p'_t}, \mathsf{t'} \rangle \end{aligned} \tag{8.33}$$

as in the definition of sample-trace*. By hypothesis, $\mathsf{Pr[t]} = \mathsf{p_t}$ implies $\mathsf{Pr[t']} = \mathsf{p'_t}$, which (by hypothesis) implies $\mathsf{Pr[t'']} = \mathsf{p''_t}$.

Case $\mathsf{idxs} = \mathsf{if\text{-}idxs\ j\ idxs}_2\ \mathsf{idxs}_3$. Let

$$\begin{aligned} \langle \mathsf{p_b}, \mathsf{b} \rangle &:= \mathsf{sample\text{-}branch}\ \{\mathsf{true}, \mathsf{false}, \bot\} \\ \mathsf{pt'} &:= \langle \mathsf{p_t} \cdot \mathsf{p_b}, \mathsf{t}[\mathsf{j} \mapsto \mathsf{b}] \rangle \end{aligned} \tag{8.34}$$

as in the definition of sample-trace*, and let $\langle \mathsf{p'_t}, \mathsf{t'} \rangle = \mathsf{pt'}$. By definition of sample-branch, $\mathsf{Pr[b]} = \mathsf{p_b}$. Because b and t are independently chosen and $\mathsf{t'}$ can be generated by no other execution path, $\mathsf{Pr[t']} = \mathsf{Pr[t]} \cdot \mathsf{Pr[b]} = \mathsf{p_t} \cdot \mathsf{p_b}$. If $\mathsf{b} = \mathsf{true}$, then in $\langle \mathsf{p''_t}, \mathsf{t''} \rangle := \mathsf{sample\text{-}trace^*\ (idxs}_2\ 0)\ \mathsf{pr'}$, by hypothesis, $\mathsf{Pr[t'']} = \mathsf{p''_t}$; similarly for $\mathsf{b} = \mathsf{false}$. □

Corollary 8.46 (sample-trace correctness)**.** *If* $\langle \mathsf{p_t}, \mathsf{t} \rangle := \mathsf{sample\text{-}trace\ idxs}$, *then* $\mathsf{Pr[t]} = \mathsf{p_t}$.

XXX: sampling from a *subset* of T_+ that induces a part of the partition of R^*:

$$\mathsf{sample\text{-}part} : \mathsf{Idxs} \Rightarrow \langle \mathbb{R}, \mathsf{Rect}\ \langle \mathsf{R}, \mathsf{T} \rangle \rangle \Rightarrow \langle \mathbb{R}, \mathsf{Rect}\ \langle \mathsf{R}, \mathsf{T} \rangle \rangle$$

$$\begin{array}{l}
\mathsf{sample\text{-}part\ idxs}\ \langle \mathsf{p}, \mathsf{R} \times \mathsf{T} \rangle\ := \\
\quad \mathsf{case}\ \ \langle \mathsf{idxs}, \langle \mathsf{p}, \mathsf{refine}\ (\mathsf{R} \times \mathsf{T}) \rangle \rangle \\
\qquad \langle \langle \rangle, \mathsf{pr} \rangle \longrightarrow \mathsf{pr} \\
\qquad \langle \langle \mathsf{idxs}_1, \mathsf{idxs}_2 \rangle, \mathsf{pr} \rangle \longrightarrow \mathsf{let}\ \ \mathsf{pr}' := \mathsf{sample\text{-}part\ idxs}_1\ \mathsf{pr} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \mathsf{sample\text{-}part\ idxs}_2\ \mathsf{pr}' \\
\qquad \langle \mathsf{if\text{-}idxs\ j\ idxs}_2\ \mathsf{idxs}_3, \langle \mathsf{p}, \mathsf{R} \times \mathsf{T} \rangle \rangle \longrightarrow \mathsf{let}\ \ \langle \mathsf{p}_\mathsf{b}, \mathsf{b} \rangle := \mathsf{sample\text{-}branch\ (proj\ j\ T)} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{pr}' := \langle \mathsf{p} \cdot \mathsf{p}_\mathsf{b}, \mathsf{R} \times \mathsf{unproj\ j\ T}\ \{\mathsf{b}\} \rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \mathsf{case}\ \ \mathsf{b} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{true} \longrightarrow \mathsf{sample\text{-}part\ (idxs}_2\ 0)\ \mathsf{pr}' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{false} \longrightarrow \mathsf{sample\text{-}part\ (idxs}_3\ 0)\ \mathsf{pr}' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \bot \longrightarrow \langle \mathsf{p} \cdot \mathsf{p}_\mathsf{b}, \varnothing \rangle
\end{array} \tag{8.35}$$

XXX: sampling a weighted point from $\mathsf{Rect}\ \langle \mathsf{R}, \mathsf{T} \rangle$:

$$\begin{array}{l}
\mathsf{let}\ \ \langle \mathsf{r}, \mathsf{q} \rangle := \mathsf{sample\text{-}point\ R} \\
\qquad\quad \mathsf{t} := \iota\, \mathsf{t} \in \mathsf{T}.\, \forall\, \mathsf{t}' \in \mathsf{T}.\, \mathsf{t} \leq \mathsf{t}' \\
\qquad\quad \mathsf{b} := \mathsf{f}\ \langle \langle \mathsf{r}, \mathsf{t} \rangle, \langle \rangle \rangle \\
\qquad\quad \mathsf{w} := \mathsf{if}\ (\mathsf{b} \in \mathsf{B})\ (1\ /\ \mathsf{p} \cdot 1\ /\ \mathsf{q})\ 0 \\
\quad \mathsf{in}\ \ \langle \mathsf{r}, \mathsf{w} \rangle
\end{array} \tag{8.36}$$

XXX: the least $\mathsf{t} \in \mathsf{T}$, or $\iota\, \mathsf{t} \in \mathsf{T}.\, \forall\, \mathsf{t}' \in \mathsf{T}.\, \mathsf{t} \leq \mathsf{t}'$, is easy to compute: set each unrestricted axis to $\bot$

XXX: decide what theorems are needed and prove them

XXX: extension: interval splitting

XXX: extension: sampling a weighted point using refinement

XXX: extension: not sampling any decision as $\bot$ when sampling in T_+; characterize the programs for which this extension doesn't terminate; explain options to detect or fix

Chapter 9

Measurability

Proving measurability is critical in proving correctness, in that it establishes that the outputs of all programs have sensible distributions. While critical, it is somewhat distracting to the main narrative. Instead of ignoring measurability, however, as is so often done, we have moved it near the end, where readers that are somehow *still starving for even more mathematics* can devour it and—possibly—finally be satiated.

9.1 Basic Definitions

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

The analogue of a topology of open sets is a σ-algebra of measurable sets.

Definition 9.1 (σ-algebra, measurable set)**.** *A collection of sets* $\mathcal{A} \subseteq \mathcal{P}\ \mathsf{X}$ *is called a* ***σ-algebra*** *on* X *if it contains* X *and is closed under complements and countable unions. The sets in* $\mathcal{A}$ *are called* ***measurable sets****.*

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

The analogue of continuity is measurability.

Definition 9.2 (measurable mapping)**.** *Let* $\mathcal{A}$ *and* $\mathcal{B}$ *be σ-algebras on* X *and* Y*. A mapping* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *is* ***$\mathcal{A}$-$\mathcal{B}$-measurable*** *if for all* $\mathsf{B} \in \mathcal{B}$, $\mathsf{preimage\ g\ B} \in \mathcal{A}$*.*

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

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

Product σ-algebras are defined analogously to product topologies.

Definition 9.3 (finite product σ-algebra)**.** *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be σ-algebras on X_1 and X_2, and define $\mathsf{X} := \mathsf{X}_1 \times \mathsf{X}_2$. The **product σ-algebra** $\mathcal{A}_1 \otimes \mathcal{A}_2$ is the smallest (i.e. coarsest) σ-algebra for which* $\mathsf{mapping\ fst\ X}$ *and* $\mathsf{mapping\ snd\ X}$ *are measurable.*

Definition 9.4 (arbitrary product σ-algebra)**.** *Let $\mathcal{A}$ be a σ-algebra on X. The **product σ-algebra** $\mathcal{A}^{\otimes \mathsf{J}}$ is the smallest σ-algebra for which, for all $\mathsf{j} \in \mathsf{J}$,* $\mathsf{mapping\ (\pi\ j)\ (J \to X)}$ *is measurable.*

9.2 Measurable Pure Computations

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

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

Definition 9.5 (measurable mapping arrow computation)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras on X and Y. A computation $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ is $\mathcal{A}$-$\mathcal{B}$-**measurable** if $\mathsf{g}\ \mathsf{A}^*$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping, where A^* is g's maximal domain.*

Theorem 9.6 (maximal domain measurability)**.** *Let $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ be an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation. Its maximal domain A^* is in $\mathcal{A}$.*

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

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

Lemma 9.7. *Let* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *be an* $\mathcal{A}$*-*$\mathcal{B}$*-measurable mapping. For any* $\mathsf{A} \in \mathcal{A}$*,* $\mathsf{restrict}\ \mathsf{g}\ \mathsf{A}$ *is* $\mathcal{A}$*-*$\mathcal{B}$*-measurable.*

Theorem 9.8. *Let* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{map}} \mathsf{Y}$ *be an* $\mathcal{A}$*-*$\mathcal{B}$*-measurable mapping arrow computation with maximal domain* A^**. For all* $\mathsf{A} \subseteq \mathsf{A}^*$ *with* $\mathsf{A} \in \mathcal{A}$*,* $\mathsf{g}\ \mathsf{A}$ *is* $\mathcal{A}$*-*$\mathcal{B}$*-measurable.*

Proof. By Theorem 7.43 (mapping arrow restriction) and Lemma 9.7. □

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

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

Lemma 9.9 (images of preimages)**.** *If* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{B} \subseteq \mathsf{Y}$*,* $\mathsf{image}\ \mathsf{g}\ (\mathsf{preimage}\ \mathsf{g}\ \mathsf{B}) \subseteq \mathsf{B}$.

Lemma 9.10 (expanded post-composition)**.** *Let* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{Y} \rightharpoonup \mathsf{Z}$ *such that* $\mathsf{range}\ \mathsf{g}_1 \subseteq \mathsf{domain}\ \mathsf{g}_2$*, and let* $\mathsf{g}_2' : \mathsf{Y} \rightharpoonup \mathsf{Z}$ *such that* $\mathsf{g}_2 \subseteq \mathsf{g}_2'$*. Then* $\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1 = \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1$.

Theorem 9.11 (mapping arrow monotonicity)**.** *Let* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{map}} \mathsf{Y}$*. For any* $\mathsf{A}' \subseteq \mathsf{A} \subseteq \mathsf{A}^*$*,* $\mathsf{g}\ \mathsf{A}' \subseteq \mathsf{g}\ \mathsf{A}$.

Proof. By Theorem 7.43 (mapping arrow restriction). □

Theorem 9.12 (maximal domain subsets)**.** *Let* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{map}} \mathsf{Y}$*. For any* $\mathsf{A} \subseteq \mathsf{A}^*$*,* $\mathsf{domain}\ (\mathsf{g}\ \mathsf{A}) = \mathsf{A}$.

Proof. Follows from Theorem 7.44. □

Now we can prove measurability.

Lemma 9.13 (($\circ_{\mathsf{map}}$) measurability)**.** *If* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *is* $\mathcal{A}$-$\mathcal{B}$*-measurable and* $\mathsf{g}_2 : \mathsf{Y} \rightharpoonup \mathsf{Z}$ *is* $\mathcal{B}$-$\mathcal{C}$*-measurable, then* $\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1$ *is* $\mathcal{A}$-$\mathcal{C}$*-measurable.*

Theorem 9.14 (($\ggg_{\mathsf{map}}$) measurability)**.** *If* $\mathsf{g}_1 : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ *is* $\mathcal{A}$-$\mathcal{B}$*-measurable and* $\mathsf{g}_2 : \mathsf{Y} \rightsquigarrow_{\mathsf{map}} \mathsf{Z}$ *is* $\mathcal{B}$-$\mathcal{C}$*-measurable, then* $\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2$ *is* $\mathcal{A}$-$\mathcal{C}$*-measurable.*

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

$$\begin{aligned}(\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A}^{**} \;=\; \mathsf{let}\ \ & \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ & \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ \mathsf{in}\ \ & \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1'\end{aligned} \tag{9.1}$$

By Theorem 9.8, g_1' is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping. Unfortunately, g_2' may not be $\mathcal{B}$-$\mathcal{C}$-measurable when $\mathsf{range}\ \mathsf{g}_1' \notin \mathcal{B}$.

Let $\mathsf{g}_2'' := \mathsf{g}_2\ \mathsf{B}^*$, which is a $\mathcal{B}$-$\mathcal{C}$-measurable mapping. By Lemma 9.13, $\mathsf{g}_2'' \circ_{\mathsf{map}} \mathsf{g}_1'$ is $\mathcal{A}$-$\mathcal{C}$-measurable. We need only show that $\mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1' = \mathsf{g}_2'' \circ_{\mathsf{map}} \mathsf{g}_1'$, which by Lemma 9.10 is true if $\mathsf{range}\ \mathsf{g}_1' \subseteq \mathsf{domain}\ \mathsf{g}_2'$ and $\mathsf{g}_2' \subseteq \mathsf{g}_2''$.

By Theorem 9.12, $\mathsf{A}^{**} \subseteq \mathsf{A}^*$ implies $\mathsf{domain}\ \mathsf{g}_1' = \mathsf{A}^{**}$. By Theorem 9.11 and Lemma 9.9,

$$\begin{aligned}\mathsf{range}\ \mathsf{g}_1' \;&=\; \mathsf{image}\ (\mathsf{g}_1\ \mathsf{A}^{**})\ (\mathsf{preimage}\ (\mathsf{g}_1\ \mathsf{A}^*)\ \mathsf{B}^*) \\ &=\; \mathsf{image}\ (\mathsf{g}_1\ \mathsf{A}^*)\ (\mathsf{preimage}\ (\mathsf{g}_1\ \mathsf{A}^*)\ \mathsf{B}^*) \\ &\subseteq\; \mathsf{B}^*\end{aligned} \tag{9.2}$$

$\mathsf{range}\ \mathsf{g}_1' \subseteq \mathsf{B}^*$ implies (by Theorem 9.12) that $\mathsf{domain}\ \mathsf{g}_2' = \mathsf{range}\ \mathsf{g}_1'$, and (by Theorem 9.11) that $\mathsf{g}_2' \subseteq \mathsf{g}_2''$. □

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

Lemma 9.15 ($\langle\cdot,\cdot\rangle_{\mathsf{map}}$ measurability)**.** *If* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *is* $\mathcal{A}$-$\mathcal{B}_1$*-measurable and* $\mathsf{g}_2 : \mathsf{X} \rightharpoonup \mathsf{Y}_2$ *is* $\mathcal{A}$-$\mathcal{B}_2$*-measurable, then* $\langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}$ *is* $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$*-measurable.*

Theorem 9.16 ($(\mathbin{\&\!\&\!\&}_{\mathsf{map}})$ measurability)**.** *If* $\mathsf{g}_1 : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}_1$ *is* $\mathcal{A}$-$\mathcal{B}_1$*-measurable and* $\mathsf{g}_2 : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}_2$ *is* $\mathcal{A}$-$\mathcal{B}_2$*-measurable, then* $\mathsf{g}_1 \mathbin{\&\!\&\!\&}_{\mathsf{map}} \mathsf{g}_2$ *is* $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$*-measurable.*

Proof. Let A_1^* and A_2^* be respectively g_1's and g_2's maximal domains. The maximal domain of $\mathsf{g}_1 \mathbin{\&\!\&\!\&}_{\mathsf{map}} \mathsf{g}_2$ is $\mathsf{A}^{**} := \mathsf{A}_1^* \cap \mathsf{A}_2^*$, which is in $\mathcal{A}$. By definition, $(\mathsf{g}_1 \mathbin{\&\!\&\!\&}_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A}^{**} = \langle \mathsf{g}_1\ \mathsf{A}^{**}, \mathsf{g}_2\ \mathsf{A}^{**} \rangle_{\mathsf{map}}$, which by Lemma 9.15 is $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$-measurable. □

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

Lemma 9.17 (union of measurable mappings)**.** *The union of a countable set of* $\mathcal{A}$-$\mathcal{B}$*-measurable mappings with disjoint domains is* $\mathcal{A}$-$\mathcal{B}$*-measurable.*

Theorem 9.18 ($\mathsf{ifte}_{\mathsf{map}}$ measurability)**.** *If* $\mathsf{g}_1 : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Bool}$*, and* $\mathsf{g}_2 : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ *and* $\mathsf{g}_3 : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ *are respectively* $\mathcal{A}$-$(\mathcal{P}\ \mathsf{Bool})$*-measurable and* $\mathcal{A}$-$\mathcal{B}$*-measurable, then* $\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3$ *is* $\mathcal{A}$-$\mathcal{B}$*-measurable.*

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

$$\begin{aligned} \mathsf{A}_2^{**} &:= \mathsf{A}_2^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{true}\} \\ \mathsf{A}_3^{**} &:= \mathsf{A}_3^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{false}\} \\ \mathsf{A}^{**} &:= \mathsf{A}_2^{**} \uplus \mathsf{A}_3^{**} \end{aligned} \tag{9.3}$$

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

$$\begin{aligned} \mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A}^{**} = \mathsf{let}\ \ & \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ & \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\}) \\ & \mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\}) \\ \mathsf{in}\ \ & \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3' \end{aligned} \tag{9.4}$$

By hypothesis, g'_1, g'_2 and g'_3 are measurable mappings. By Theorem 7.43 (mapping arrow restriction), g'_2 and g'_3 have disjoint domains. Apply Lemma 9.17. □

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

Theorem 9.19 (measurability of $\varnothing$)**.** *For any σ-algebras $\mathcal{A}$ and $\mathcal{B}$, the empty mapping $\varnothing$ is $\mathcal{A}$-$\mathcal{B}$-measurable.*

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

Theorem 9.20 (measurability under $\mathsf{lazy_{map}}$)**.** *Let* $\mathsf{g} : 1 \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})$*. If* $\mathsf{g}\ 0$ *is $\mathcal{A}$-$\mathcal{B}$-measurable, then* $\mathsf{lazy_{map}}\ \mathsf{g}$ *is $\mathcal{A}$-$\mathcal{B}$-measurable.*

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

$$\mathsf{lazy_{map}}\ \mathsf{g}\ \mathsf{A}^{**} \ = \ \mathsf{if}\ (\mathsf{A}^{**} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}^{**}) \tag{9.5}$$

If $\mathsf{A}^{**} = \varnothing$, then $\mathsf{lazy_{map}}\ \mathsf{g}\ \mathsf{A}^{**} = \varnothing$; apply Theorem 9.19. If $\mathsf{A}^{**} \neq \varnothing$, then $\mathsf{lazy_{map}}\ \mathsf{g} = \mathsf{g}\ 0$, which is $\mathcal{A}$-$\mathcal{B}$-measurable. □

9.3 Measurable Probabilistic Computations

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

Definition 9.21 (measurable mapping* arrow computation)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras on* $(\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$ *and* Y*. A computation* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ *is $\mathcal{A}$-$\mathcal{B}$-**measurable** if* $\mathsf{g}\ \mathsf{j}_0$ *is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation.*

Theorem 9.22. *If* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ *is $\mathcal{A}$-$\mathcal{B}$-measurable, then for all* $\mathsf{j} \in \mathsf{J}$*,* $\mathsf{g}\ \mathsf{j}$ *is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation.*

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

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

Definition 9.23 (standard σ-algebra)**.** *For a set* X *used as a type,* $\Sigma\ \mathsf{X}$ *denotes its* ***standard σ-algebra****, which must be defined under the following constraints:*

$$\Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle = \Sigma\ \mathsf{X}_1 \otimes \Sigma\ \mathsf{X}_2 \tag{9.6}$$

$$\Sigma\ (\mathsf{J} \to \mathsf{X}) = (\Sigma\ \mathsf{X})^{\otimes \mathsf{J}} \tag{9.7}$$

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:

$$\Sigma\ \mathsf{Bool}\ ::=\ \mathcal{P}\ \mathsf{Bool} \tag{9.8}$$

$$\Sigma\ \mathsf{T}\ ::=\ \mathcal{P}\ \mathsf{T} \tag{9.9}$$

Lemma 9.24 (measurable mapping arrow lifts)**.** $\mathsf{arr}_{\mathsf{map}}\ \mathsf{id}$*,* $\mathsf{arr}_{\mathsf{map}}\ \mathsf{fst}$ *and* $\mathsf{arr}_{\mathsf{map}}\ \mathsf{snd}$ *are measurable.* $\mathsf{arr}_{\mathsf{map}}\ (\mathsf{const}\ \mathsf{b})$ *is measurable if* $\{\mathsf{b}\}$ *is a measurable set. For all* $\mathsf{j} \in \mathsf{J}$*,* $\mathsf{arr}_{\mathsf{map}}\ (\pi\ \mathsf{j})$ *is measurable.*

Corollary 9.25 (measurable mapping* arrow lifts)**.** $\mathsf{arr}_{\mathsf{map}^*}\ \mathsf{id}$*,* $\mathsf{arr}_{\mathsf{map}^*}\ \mathsf{fst}$ *and* $\mathsf{arr}_{\mathsf{map}^*}\ \mathsf{snd}$ *are measurable.* $\mathsf{arr}_{\mathsf{map}^*}\ (\mathsf{const}\ \mathsf{b})$ *is measurable if* $\{\mathsf{b}\}$ *is a measurable set.* $\mathsf{random}_{\mathsf{map}^*}$ *and* $\mathsf{branch}_{\mathsf{map}^*}$ *are measurable.*

Theorem 9.26 (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 $\mathsf{arr}_{\mathsf{map}}\ \mathsf{fst}$ and $\mathsf{arr}_{\mathsf{map}}\ \mathsf{snd}$. □

Theorem 9.27 ($\mathsf{ifte}^{\Downarrow}_{\mathsf{map}^*}$ measurability)**.** $\mathsf{ifte}^{\Downarrow}_{\mathsf{map}^*}$ *is measurable.*

Proof. $\mathsf{branch}_{\mathsf{map}^*}$ is measurable, and $\mathsf{arr}_{\mathsf{map}}\ \mathsf{agrees}$ is measurable by (9.8). □

We can now prove all nonrecursive programs measurable by induction.

Definition 9.28 (finite expression)**.** *A **finite expression** is any expression for which no subexpression is a first-order application.*

Theorem 9.29 (all finite expressions are measurable)**.** *For all finite expressions e, $[\![e]\!]_{\mathsf{map}^*}$ is measurable.*

Proof. By structural induction and the above theorems. □

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

Theorem 9.30 (approximation with finite expressions)**.** *Let $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map}^*} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ and $\mathsf{t} \in \mathsf{T}$. Define $\mathsf{A} := (\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}$. There is a finite expression e' for which $[\![e']\!]_{\mathsf{map}^*}\ \mathsf{j}_0\ \mathsf{A} = \mathsf{g}\ \mathsf{j}_0\ \mathsf{A}$.*

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

Theorem 9.31 (all probabilistic expressions are measurable)**.** *For all expressions e, $[\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ is measurable.*

Proof. Let $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ and $\mathsf{g}' := \mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \mathsf{T}) \times \mathsf{X})$. By Corollary 7.50 (correct computation everywhere), $\mathsf{g}' = \mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*$ where A^* is g's maximal domain; thus we need only show g' is a measurable mapping.

By Theorem 7.43 (mapping arrow restriction),

$$\mathsf{g}' \;=\; \bigcup_{\mathsf{t} \in \mathsf{T}} \mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}) \tag{9.10}$$

By Theorem 9.30 (approximation with finite expressions), for every $\mathsf{t} \in \mathsf{T}$, there is a finite expression whose interpretation agrees with g on $(\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}$. Therefore, by Theorem 9.29 (all finite expressions are measurable), $\mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X})$ is a measurable mapping. By Theorem 7.43 (mapping arrow restriction), they have disjoint domains. By Lemma 9.17 (union of measurable mappings), their union is measurable. □

Theorem 9.31 remains true when $[\![\cdot]\!]_{\mathsf{a}}$ is extended with any rule whose right side is measurable, including rules for real arithmetic, equality, inequality and limits. More generally, any continuous or (countably) piecewise continuous function can be made available as a language primitive, as long as its domain's and codomain's standard σ-algebras are generated from their topologies.

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

9.4 Measurable Projections

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

$$\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}\ (\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*)\ \mathsf{B})) \tag{9.11}$$

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

Theorem 9.32 (measurable finite projections)**.** *Let* $\mathsf{A} \in \Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$. *If* X_2 *is at most countable and* $\Sigma\ \mathsf{X}_2 = \mathcal{P}\ \mathsf{X}_2$*, then* $\mathsf{image\ fst\ A} \in \mathcal{A}_1$.

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

Theorem 9.33. *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ *be measurable. If* X *is at most countable and* $\Sigma\ \mathsf{X} = \mathcal{P}\ \mathsf{X}$*, for all* $\mathsf{B} \in \Sigma\ \mathsf{Y}$*,* $\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}\ (\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*)\ \mathsf{B}) \in \Sigma\ \mathsf{R}$.

Proof. T is countable and $\Sigma\ \mathsf{T} = \mathcal{P}\ \mathsf{T}$ by (9.9); apply Theorem 9.32 twice. □

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

Chapter 10

Sampling Algorithm Proofs

XXX: You're still here?

10.1 Basic Definitions

XXX: unlike preceeding chapter, basic definitions not meant to be self-contained, but to 1) define things typically regarded as "just syntax" or "just formalism" as λ-calculus functions so we can manipulate them more mechanically; 2) give interested readers terminology to look up; 3) give precise definitions for ambiguous terms (especially "uniformly σ-finite")

Definition 10.1 (almost-everywhere equality)**.** *A measurable predicate* $\mathsf{p?}$ *holds* ***almost everywhere*** *with respect to measure* m *when it holds on the* complement *of a measure-zero set:*

$$\begin{aligned} &\mathsf{ae?} : (\mathsf{Set\ X} \rightharpoonup [0, +\infty]) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Bool}) \Rightarrow \mathsf{Bool} \\ &\mathsf{ae?\ m\ p?} \ := \ \mathsf{m\ (preimage\ p?\ \{false\})} = 0 \end{aligned} \tag{10.1}$$

Similarly, two total mappings are equal almost everywhere (with respect to a measure m*) when they are equal on all but a measure-zero set:*

$$\begin{aligned} &\mathsf{ae\text{-}equal?} : (\mathsf{Set\ X} \rightharpoonup [0, +\infty]) \Rightarrow (\mathsf{X} \to \mathsf{Y}) \Rightarrow (\mathsf{X} \to \mathsf{Y}) \Rightarrow \mathsf{Bool} \\ &\mathsf{ae\text{-}equal?\ m\ g_1\ g_2} \ := \ \mathsf{ae?\ m\ } \lambda \mathsf{a} \in \mathsf{domain\ g_1}.\, \mathsf{g_1\ a} = \mathsf{g_2\ a} \end{aligned} \tag{10.2}$$

From here on, we write "$\mathsf{g_1} = \mathsf{g_2}$ (m-a.e.)" instead of $\mathsf{ae\text{-}equal?\ m\ g_1\ g_2}$.

[XXX: define σ-finite]

Lebesgue[1] integration:

$$\begin{aligned} &\mathsf{int} : (\mathsf{X} \to \mathbb{R}) \Rightarrow (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \Rightarrow (\mathsf{Set}\ \mathsf{X} \rightharpoonup [-\infty, +\infty]) \\ &\mathsf{int}\ \mathsf{f}\ \mathsf{m}\ \mathsf{A} \ = \ \int_{\mathsf{A}} \mathsf{f}\ d\mathsf{m} \end{aligned} \tag{10.3}$$

XXX: λ_{ZFC}-definable, but its definition does not give extra clarity

XXX: Lebesgue integration is awesome as a lambda because indefinite integration, which yields a *measure*, can be done simply by partial application; will demonstrate this shortly in the statement of Lemma 10.3

Lemma 10.2 (indefinite integration yields measures)**.** *If* $\mathsf{f} : \mathsf{X} \to [0, +\infty)$ *is measurable and* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *is a measure, then* $\mathsf{int}\ \mathsf{f}\ \mathsf{m}$ *is a measure. [XXX: preservation of measure properties like* σ*-finite?]*

For real-valued functions, Lebesgue integration gives another, sometimes more convenient way to characterize almost-everywhere equality: two functions are equal almost everywhere if and only if their indefinite integrals are equal.

Lemma 10.3 (real function a.e. equality)**.** *If* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *is a* σ*-finite measure and* $\mathsf{f}_1, \mathsf{f}_2 : \mathsf{X} \to \mathbb{R}$ *are measurable, then* $\mathsf{f}_1 = \mathsf{f}_2$ *(*m*-a.e) if and only if* $\mathsf{int}\ \mathsf{f}_1\ \mathsf{m} = \mathsf{int}\ \mathsf{f}_2\ \mathsf{m}$*.*

In differential calculus (hereafter simply "calculus"), indefinite integration has an inverse: differentiation. In measure theory, indefinite Lebesgue integration also has an inverse, which is also called differentiation. In calculus, differentiation is defined only for differentiable functions. In measure theory, the analogous property is called absolute continuity.

Definition 10.4 (absolute continuity)**.** *Given measures* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$*,* m_1 *is* ***absolutely continuous*** *with respect to* m_2 *if* $\mathsf{m}_1 \ll \mathsf{m}_2$*, where*

$$\begin{aligned} &(\ll) : (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \Rightarrow (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \Rightarrow \mathsf{Bool} \\ &\mathsf{m}_1 \ll \mathsf{m}_2 \ := \ \forall\, \mathsf{A} \in \mathsf{domain}\ \mathsf{m}_2.\, \mathsf{m}_2\ \mathsf{A} = 0 \implies \mathsf{m}_1\ \mathsf{A} = 0 \end{aligned} \tag{10.4}$$

[1] Pronounced "lehBEG," and named after French mathematician Henri Lebesgue.

By Definition 10.4, "$\mathsf{m}_1 \ll \mathsf{m}_2$" means that m_1 has at least as many measure-zero sets as m_2, and is therefore, in a sense, smaller.

Because we are doing probability, we need to deal only with nonnegative derivatives, for which operations have simpler closure conditions. The types of nonnegative integration and differentiation are

$$\begin{aligned} \mathsf{int}^+ &: (\mathsf{X} \to [0, +\infty)) \Rightarrow (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \Rightarrow (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \\ \mathsf{diff}^+ &: (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \Rightarrow (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \Rightarrow (\mathsf{X} \to [0, +\infty)) \end{aligned} \tag{10.5}$$

The function diff^+ returns a **Radon-Nikodým derivative**. Such derivatives are named after the following theorem, which gives circumstances under which $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$ exists, and states that int^+ is the left inverse of diff^+ (with second arguments held constant).

Lemma 10.5 (Radon-Nikodým theorem)**.** *If* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *are* σ*-finite measures and* $\mathsf{m}_1 \ll \mathsf{m}_2$*, then* $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$ *exists and* $\mathsf{m}_1 = \mathsf{int}^+\ (\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2)\ \mathsf{m}_2$.

The function $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2 : \mathsf{X} \to [0, +\infty)$ is often called the **density** of m_1 with respect to m_2. ("With respect to m_2" is usually not stated when m_2 is Lebesgue measure—i.e. length, area, or volume—as with the densities of most common probability distributions.) By Lemma 10.3 (almost-everywhere equality of real functions), $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$ is unique up to equality m_2-a.e.

By analogy to 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 10.6. *If* $\mathsf{f}_1 : \mathsf{X} \to [0, +\infty)$ *is measurable and* $\mathsf{m}_2 : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *is a* σ*-finite measure, then* $\mathsf{int}^+\ \mathsf{f}_1\ \mathsf{m}_2 \ll \mathsf{m}_2$ *and* $\mathsf{f}_1 = \mathsf{diff}^+\ (\mathsf{int}^+\ \mathsf{f}_1\ \mathsf{m}_2)\ \mathsf{m}_2$ *(*m_2*-a.e.).*

The previous two theorems are analogous to the two parts of the fundamental theorem of calculus. The next two theorems are analogous to others in calculus: that Radon-Nikodým differentiation is linear in its first argument.

Lemma 10.7. *Let* $\mathsf{m}_1, \mathsf{m}_2, \mathsf{m} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *be* σ*-finite measures with* $\mathsf{m}_1 \ll \mathsf{m}$ *and* $\mathsf{m}_2 \ll \mathsf{m}$. *Then* $\mathsf{m}_1 + \mathsf{m}_2 \ll \mathsf{m}$ *and* $\mathsf{diff}^+\ (\mathsf{m}_1 + \mathsf{m}_2)\ \mathsf{m}\ =\ \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m} + \mathsf{diff}^+\ \mathsf{m}_2\ \mathsf{m}$ *(*m*-a.e.).*

Lemma 10.8. *Let* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *be* σ*-finite measures with* $\mathsf{m}_1 \ll \mathsf{m}_2$. *For all* $\alpha \geq 0$ *and* $\beta > 0$, $\alpha \cdot \mathsf{m}_1 \ll \beta \cdot \mathsf{m}_2$ *and* $\mathsf{diff}^+\ (\alpha \cdot \mathsf{m}_1)\ (\beta \cdot \mathsf{m}_2)\ =\ \dfrac{\alpha}{\beta} \cdot \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$ *(*m*-a.e.).*

As in differentiation in calculus, there is a chain rule for diff^+:

Lemma 10.9. *Let* $\mathsf{m}_1, \mathsf{m}_2, \mathsf{m}_3 : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *be* σ*-finite measures with* $\mathsf{m}_1 \ll \mathsf{m}_2$ *and* $\mathsf{m}_2 \ll \mathsf{m}_3$. *Then* $\mathsf{m}_1 \ll \mathsf{m}_3$ *and* $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_3\ =\ \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2 \cdot \mathsf{diff}^+\ \mathsf{m}_2\ \mathsf{m}_3$ *(*m_3*-a.e.).*

Now we finally get to rules for which there is no analogy in calculus. First, a rule for reciprocals [XXX: probably don't need this—double-check]:

Lemma 10.10. *Let* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *be* σ*-finite measures with* $\mathsf{m}_2 \ll \mathsf{m}_1$ *and* $\mathsf{m}_1 \ll \mathsf{m}_2$. *Then* $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ =\ 1\ /\ \mathsf{diff}^+\ \mathsf{m}_2\ \mathsf{m}_1$ *(*m_1*-a.e. or* m_2*-a.e.).*

Second, a way to integrate out densities, or to use differentiation to change the base measure in Lebesgue integration:

Lemma 10.11 (change of measure)**.** *Let* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *be* σ*-finite measures with* $\mathsf{m}_1 \ll \mathsf{m}_2$, *and* $\mathsf{g} : \mathsf{X} \to \mathbb{R}$ *be measurable. Then* $\mathsf{int}\ \mathsf{g}\ \mathsf{m}_1\ =\ \mathsf{int}\ (\mathsf{g} \cdot \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2)\ \mathsf{m}_2$.

XXX: intro the following; something about composing two processes to yield another, where the second depends on the outcome of the first

Definition 10.12 (transition kernel)**.** *A function* $\mathsf{k} : \mathsf{X} \to \mathsf{Set}\ \mathsf{Y} \rightharpoonup [0, +\infty]$ *is a* ***transition kernel*** *when both of the following hold.*

- *For all* $\mathsf{a} \in \mathsf{X}$, $\mathsf{k}\ \mathsf{a}$ *is a measure.*
- *For all* $\mathsf{B} \in \Sigma\ \mathsf{Y}$, $\lambda \mathsf{a} \in \mathsf{X}.\, \mathsf{k}\ \mathsf{a}\ \mathsf{B}$ *is measurable.*

For any measure property P, *we say* k *has property* P *when for all* $\mathsf{a} \in \mathsf{X}$, $\mathsf{P}\ (\mathsf{k}\ \mathsf{a})$. *[XXX: won't need this last statement if I go with uniformly* σ*-finite]*

Definition 10.13 (uniformly σ-finite)**.** *A transition kernel* $\mathsf{k} : \mathsf{X} \to \mathsf{Set}\ \mathsf{Y} \rightharpoonup [0, +\mathsf{infty}]$ *is* ***uniformly σ-finite*** *when there exists an at-most-countable partition* $\mathsf{s} : \mathbb{N} \to \mathsf{Set}\ \mathsf{Y}$ *such that* $\mathsf{k}\ \mathsf{a}\ (\mathsf{s}\ \mathsf{n}) < +\infty$ *for all* $\mathsf{a} \in \mathsf{X}$ *and* $\mathsf{n} \in \mathbb{N}$*.*

Lemma 10.14 (finite transition kernel products)**.** *Let* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \to [0, +\infty]$ *be a σ-finite measure and* $\mathsf{k} : \mathsf{X} \to \mathsf{Set}\ \mathsf{Y} \rightharpoonup [0, +\infty]$ *be a uniformly σ-finite transition kernel. There exists a unique σ-finite measure* $\mathsf{m} \times \mathsf{k} : \mathsf{Set}\ \langle \mathsf{X}, \mathsf{Y} \rangle \rightharpoonup [0, +\infty]$ *defined by extending* $(\mathsf{m} \times \mathsf{k})\ (\mathsf{A} \times \mathsf{B}) = \mathsf{int}^+\ (\lambda\, \mathsf{a} \in \mathsf{X}.\, \mathsf{k}\ \mathsf{a}\ \mathsf{B})\ \mathsf{m}\ \mathsf{A}$ *to a product measure.*

Lemma 10.15 (Fubini's for transition kernels)**.** *Let* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \to [0, +\infty]$ *be a σ-finite measure and* $\mathsf{k} : \mathsf{X} \to \mathsf{Set}\ \mathsf{Y} \rightharpoonup [0, +\infty]$ *be a uniformly σ-finite transition kernel. If* $\mathsf{f} : \mathsf{X} \times \mathsf{Y} \to [-\infty, +\infty]$ *is measurable, and nonnegative or* $(\mathsf{m} \times \mathsf{k})$*-integrable, then*

$$\begin{aligned} &\mathsf{int}\ \mathsf{f}\ (\mathsf{m} \times \mathsf{k})\ (\mathsf{X} \times \mathsf{Y}) \\ &\quad = \ \mathsf{int}\ (\lambda\, \mathsf{a} \in \mathsf{X}.\, \mathsf{int}\ (\lambda\, \mathsf{b} \in \mathsf{Y}.\, \mathsf{f}\ \langle \mathsf{a}, \mathsf{b} \rangle)\ (\mathsf{k}\ \mathsf{a})\ \mathsf{Y})\ \mathsf{m}\ \mathsf{X} \end{aligned} \tag{10.6}$$

10.2 Sampling Proofs

Theorem 10.16 (partitioned importance sampling correctness)**.** *Let* X*,* P*,* N*,* s*,* p*, and* Q *as in Definition 8.27 (partitioned importance sampling) such that* $\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}) \ll \mathsf{Q}\ \mathsf{n}$ *for all* $\mathsf{n} \in \mathsf{N}$*. Define* $\mathsf{P_N} : \mathsf{Set}\ \mathsf{N} \to [0, 1]$ *by extending* p *to a measure.*

If $\mathsf{g} : \mathsf{X} \to \mathbb{R}$ *is a* P*-integrable mapping, and*

$$\begin{aligned} &\mathsf{g}' : \mathsf{N} \times \mathsf{X} \to \mathbb{R} \\ &\mathsf{g}'\ \langle \mathsf{n}, \mathsf{a} \rangle \ := \ \mathsf{g}\ \mathsf{a} \cdot \frac{1}{\mathsf{p}\ \mathsf{n}} \cdot \mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{a} \end{aligned} \tag{10.7}$$

then $\mathsf{int}\ \mathsf{g}'\ (\mathsf{P_N} \times \mathsf{Q})\ (\mathsf{N} \times \mathsf{X}) \ = \ \mathsf{int}\ \mathsf{g}\ \mathsf{P}\ \mathsf{X}$*.*

Proof. Expand g' in the left-hand side of the equality, apply Lemma 10.15 (Fubini's for transition kernels), rearrange inside terms, and apply Lemma 10.11 (change of measure) to

cancel Q n:

$$\begin{aligned}
&\mathsf{int}\ \mathsf{g}'\ (\mathsf{P_N} \times \mathsf{Q})\ (\mathsf{N} \times \mathsf{X}) \\
&\quad = \mathsf{int}\ (\lambda\langle \mathsf{n}, \mathsf{a}\rangle \in \mathsf{N} \times \mathsf{X}.\, \mathsf{g}\ \mathsf{a} \cdot \frac{1}{\mathsf{p}\ \mathsf{n}} \cdot \mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{a})\ (\mathsf{P_N} \times \mathsf{Q})\ (\mathsf{N} \times \mathsf{X}) \\
&\quad = \mathsf{int}\ (\lambda \mathsf{n} \in \mathsf{N}.\, \mathsf{int}\ (\lambda \mathsf{a} \in \mathsf{X}.\, \mathsf{g}\ \mathsf{a} \cdot \frac{1}{\mathsf{p}\ \mathsf{n}} \cdot \mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{a})\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{X})\ \mathsf{P_N}\ \mathsf{N} \\
&\quad = \mathsf{int}\ (\lambda \mathsf{n} \in \mathsf{N}.\, \mathsf{int}\ (\mathsf{g} \cdot \frac{1}{\mathsf{p}\ \mathsf{n}} \cdot \mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ (\mathsf{Q}\ \mathsf{n}))\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{X})\ \mathsf{P_N}\ \mathsf{N} \\
&\quad = \mathsf{int}\ (\lambda \mathsf{n} \in \mathsf{N}.\, \mathsf{int}\ (\mathsf{g} \cdot \frac{1}{\mathsf{p}\ \mathsf{n}})\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ \mathsf{X})\ \mathsf{P_N}\ \mathsf{N}
\end{aligned}$$

Because N is at most countable, turn integration into summation; then use σ-additivity of measures:

$$\begin{aligned}
&= \sum_{\mathsf{n} \in \mathsf{N}} \mathsf{p}\ \mathsf{n} \cdot \mathsf{int}\ (\mathsf{g} \cdot \frac{1}{\mathsf{p}\ \mathsf{n}})\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ \mathsf{X} \\
&= \sum_{\mathsf{n} \in \mathsf{N}} \mathsf{int}\ \mathsf{g}\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ \mathsf{X} \\
&= \sum_{\mathsf{n} \in \mathsf{N}} \mathsf{int}\ \mathsf{g}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}) \\
&= \mathsf{int}\ \mathsf{g}\ \mathsf{P}\ (\textstyle\bigcup_{\mathsf{n} \in \mathsf{N}}\ (\mathsf{s}\ \mathsf{n})) \\
&= \mathsf{int}\ \mathsf{g}\ \mathsf{P}\ \mathsf{X}
\end{aligned}$$

□

Theorem 10.17. *Let* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *be* σ*-finite measures such that* $\mathsf{m}_1 \ll \mathsf{m}_2$*, and* $\mathsf{k} : \mathsf{X} \to \mathsf{Set}\ \mathsf{Y} \rightharpoonup [0, +\infty]$ *be a uniformly* σ*-finite transition kernel. Then* $\mathsf{m}_1 \times \mathsf{k} \ll \mathsf{m}_2 \times \mathsf{k}$ *and* $\mathsf{diff}^+\ (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{m}_2 \times \mathsf{k}) = (\lambda\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{X} \times \mathsf{Y}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a})$ *(*$\mathsf{m}_2 \times \mathsf{k}$*-a.e.).*

Proof. XXX: prove absolute continuity

By Lemma 10.3, they are equal $\mathsf{m}_2 \times \mathsf{k}$-a.e. if and only if their indefinite integrals w.r.t. $\mathsf{m}_2 \times \mathsf{k}$ are equal.

Let A and B be measurable subsets of X and Y respectively. Starting from the left-hand

side, apply Lemma 10.11 (change of measure):

$$\begin{aligned}
&\mathsf{int}\ (\mathsf{diff}^+\ (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{m}_2 \times \mathsf{k}))\ (\mathsf{m}_2 \times \mathsf{k})\ (\mathsf{A} \times \mathsf{B}) \\
&\quad = \ \mathsf{int}\ (\lambda\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{X} \times \mathsf{Y}.\, 1)\ (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{A} \times \mathsf{B}) \\
&\quad = \ (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{A} \times \mathsf{B})
\end{aligned} \tag{10.8}$$

From the right-hand side, apply Lemma 10.15 (Fubini's for transition kernels), rearrange terms, apply Lemma 10.11 (change of measure), and Lemma 10.14 (finite transition kernel products):

$$\begin{aligned}
&\mathsf{int}\ (\lambda\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{X} \times \mathsf{Y}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a})\ (\mathsf{m}_2 \times \mathsf{k})\ (\mathsf{A} \times \mathsf{B}) \\
&\quad = \ \mathsf{int}\ (\lambda \mathsf{a} \in \mathsf{X}.\, \mathsf{int}\ (\lambda \mathsf{b} \in \mathsf{Y}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a})\ (\mathsf{k}\ \mathsf{a})\ \mathsf{B})\ \mathsf{m}_2\ \mathsf{A} \\
&\quad = \ \mathsf{int}\ (\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2 \cdot \lambda \mathsf{a} \in \mathsf{X}.\, \mathsf{int}\ (\lambda \mathsf{b} \in \mathsf{Y}.\, 1)\ (\mathsf{k}\ \mathsf{a})\ \mathsf{B})\ \mathsf{m}_2\ \mathsf{A} \\
&\quad = \ \mathsf{int}\ (\lambda \mathsf{a} \in \mathsf{X}.\, \mathsf{k}\ \mathsf{a}\ \mathsf{B})\ \mathsf{m}_1\ \mathsf{A} \\
&\quad = \ (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{A} \times \mathsf{B})
\end{aligned} \tag{10.9}$$

By uniqueness of σ-finite product measures, the integrals are equal. □

It is not hard to extend the preceeding theorem to arbitrary sublists of finite lists, or to arbitrary finite substructures of any algebraic data type, by finite induction. But we need a version of it for arbitrary finite substructures of infinite binary trees, which we have defined non-inductively as mappings $\mathsf{R} := \mathsf{J} \to [0, 1]$ from tree indexes to reals.

The main idea is to define a transformation from any $\mathsf{r} \in \mathsf{R}$ to a pair $\langle \mathsf{r}_{\mathsf{fin}}, \mathsf{r}_{\mathsf{inf}}\rangle$, where $\mathsf{r}_{\mathsf{fin}}$ is a finite substructure and $\mathsf{r}_{\mathsf{inf}}$ is the rest of it, and apply Theorem [XXX: preceeding]. The proof is easier to do abstractly: without specifying the structure of r, without requiring the substructure to be finite, and without requiring the transformation to be a projection.

Theorem 10.18. *Let* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *be* σ*-finite measures such that* $\mathsf{m}_1 \ll \mathsf{m}_2$*, and* $\mathsf{k} : \mathsf{X} \to \mathsf{Set}\ \mathsf{Y} \rightharpoonup [0, +\infty]$ *be a uniformly* σ*-finite transition kernel. Let* $\mathsf{f} : \Omega \to \mathsf{X} \times \mathsf{Y}$ *be*

measurable and invertible, and let $\mu_1, \mu_2 : \mathsf{Set}\ \Omega \rightharpoonup [0,1]$ *so that*

$$\begin{aligned} \mu_1\ \Omega' &= (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{image}\ \mathsf{f}\ \Omega') \\ \mu_2\ \Omega' &= (\mathsf{m}_2 \times \mathsf{k})\ (\mathsf{image}\ \mathsf{f}\ \Omega') \end{aligned} \tag{10.10}$$

Then $\mathsf{diff}^+\ \mu_1\ \mu_2 = \lambda\omega \in \Omega.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ (\mathsf{fst}\ (\mathsf{f}\ \omega))$ *(μ_2-a.e.).*

Proof. Suppose $\Omega' \subseteq \Omega$ is measurable and let $\mathsf{C} := \mathsf{image}\ \mathsf{f}\ \Omega'$. Let f^{-1} be the inverse of f. [XXX: justify steps]

$$\begin{aligned} &\mathsf{int}\ (\mathsf{diff}^+\ \mu_1\ \mu_2)\ \mu_2\ \Omega' \\ &\quad= \mathsf{int}\ (1_{\Omega'} \cdot \mathsf{diff}^+\ \mu_1\ \mu_2)\ \mu_2\ \Omega \\ &\quad= \mathsf{int}\ 1_{\Omega'}\ \mu_1\ \Omega \\ &\quad= \mathsf{int}\ (1_{\Omega'} \circ_{\mathsf{map}} \mathsf{f}^{-1})\ (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{X} \times \mathsf{Y}) \\ &\quad= \mathsf{int}\ 1_{\mathsf{C}}\ (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{X} \times \mathsf{Y}) \\ &\quad= \mathsf{int}\ (1_{\mathsf{C}} \cdot \mathsf{diff}^+\ (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{m}_2 \times \mathsf{k}))\ (\mathsf{m}_2 \times \mathsf{k})\ (\mathsf{X} \times \mathsf{Y}) \\ &\quad= \mathsf{int}\ (1_{\mathsf{C}} \cdot \lambda\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{X} \times \mathsf{Y}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a})\ (\mathsf{m}_2 \times \mathsf{k})\ (\mathsf{X} \times \mathsf{Y}) \\ &\quad= \mathsf{int}\ ((1_{\mathsf{C}} \cdot \lambda\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{X} \times \mathsf{Y}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a}) \circ_{\mathsf{map}} \mathsf{f})\ \mu_2\ \Omega \\ &\quad= \mathsf{int}\ (1_{\Omega'} \cdot (\lambda\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{X} \times \mathsf{Y}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a}) \circ_{\mathsf{map}} \mathsf{f})\ \mu_2\ \Omega \\ &\quad= \mathsf{int}\ ((\lambda\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{X} \times \mathsf{Y}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a}) \circ_{\mathsf{map}} \mathsf{f})\ \mu_2\ \Omega' \\ &\quad= \mathsf{int}\ (\lambda\omega \in \Omega.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ (\mathsf{fst}\ (\mathsf{f}\ \omega)))\ \mu_2\ \Omega' \end{aligned}$$

Apply Lemma 10.3 (real function a.e. equality). □

Thus, two measures μ_1 and μ_2 on infinite structures that can be decomposed into products $\mathsf{m}_1 \times \mathsf{k}$ and $\mathsf{m}_2 \times \mathsf{k}$ such that $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$ exists have a Radon-Nikodým derivative that can be completely characterized in terms of $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$.

Application to infinite binary trees merely requires defining the transformation f, which is clearly invertible.

Corollary 10.19. *Let* $\mathsf{J}' \subseteq \mathsf{J}$*, let* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set}\ (\mathsf{J}' \to [0,1]) \rightharpoonup [0,1]$ *be* σ*-finite measures such that* $\mathsf{m}_1 \ll \mathsf{m}_2$*, and let* $\mathsf{k} : (\mathsf{J}' \to [0,1]) \to (\mathsf{Set}\ (\mathsf{J} \backslash \mathsf{J}' \to [0,1]) \rightharpoonup [0,1])$ *be a uniformly* σ*-finite transition kernel. Let*

$$\begin{aligned} \mathsf{f} &: \mathsf{R} \to (\mathsf{J}' \to [0,1]) \times (\mathsf{J} \backslash \mathsf{J}' \to [0,1]) \\ \mathsf{f}\ \mathsf{r} &:= \langle \mathsf{restrict}\ \mathsf{r}\ \mathsf{J}', \mathsf{restrict}\ \mathsf{r}\ (\mathsf{J} \backslash \mathsf{J}') \rangle \end{aligned} \tag{10.11}$$

and let $\mu_1, \mu_2 : \mathsf{Set}\ \mathsf{R} \rightharpoonup [0,1]$ *so that*

$$\begin{aligned} \mu_1\ \mathsf{R}' &= (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{image}\ \mathsf{f}\ \mathsf{R}') \\ \mu_2\ \mathsf{R}' &= (\mathsf{m}_2 \times \mathsf{k})\ (\mathsf{image}\ \mathsf{f}\ \mathsf{R}') \end{aligned} \tag{10.12}$$

Then $\mathsf{diff}^+\ \mu_1\ \mu_2 = \lambda \mathsf{r} \in \mathsf{R}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ (\mathsf{restrict}\ \mathsf{r}\ \mathsf{J}')$ *(μ_2-a.e.).*

Chapter 11

Related Work

References

[1] Haskell 98 language and libraries, the revised report, December 2002. URL `http://www.haskell.org/onlinereport/`.

[2] Stephen Abbott. *Understanding Analysis*. Springer, 2001.

[3] Peter Aczel. An introduction to inductive definitions. *Studies in Logic and the Foundations of Mathematics*, 90:739–782, 1977.

[4] G. Amato and F. Scozzari. The abstract domain of parallelotopes. *Electronic Notes in Theoretical Computer Science*, 287:17–28, November 2012.

[5] Robert J. Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5:614–630, 1961.

[6] Bruno Barras. Sets in Coq, Coq in sets. *Journal of Formalized Reasoning*, 3(1), 2010.

[7] C. Berline and K. Grue. A κ-denotational semantics for Map Theory in ZFC+SI. *Theoretical Computer Science*, 179(1–2):137–202, 1997.

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004. URL `http://www.labri.fr/publications/l3a/2004/BC04`.

[9] Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio Russo. Deriving probability density functions from probabilistic functional programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2013.

[10] Keith A Bonawitz. *Composable Probabilistic Inference with Blaise*. PhD thesis, Massachusetts Institute of Technology, 2008.

[11] Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for Bayesian machine learning. In *European Symposium on Programming*, pages 77–96, 2011.

[12] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Principles of Programming Languages*, pages 1–13, 2005.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[14] Ryan Culpepper. *Refining Syntactic Sugar: Tools for Supporting Macro Development*. PhD thesis, Northeastern University, 2010. To Appear.

[15] M.H. DeGroot and M.J. Schervish. *Probability and Statistics*. Addison Wesley Publishing Company, Inc., 2012. ISBN 9780321500465.

[16] R. C. Flagg and J. Myhill. A type-free system extending ZFC. *Annals of Pure and Applied Logic*, 43:79–97, 1989.

[17] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[18] Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence*, 2008.

[19] Michael Hanus. Multi-paradigm declarative languages. In *Logic Programming*, pages 45–75. 2007.

[20] Martin Hofmann, Benjamin C. Pierce, , and Daniel Wagner. Edit lenses. In *Principles of Programming Languages*, 2012.

[21] K. Hrbacek and T.J. Jech. *Introduction to set theory*. Pure and Applied Mathematics. M. Dekker, 1999.

[22] John Hughes. Generalizing monads to arrows. In *Science of Computer Programming*, volume 37, pages 67–111, 2000.

[23] Joe Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002.

[24] Claire Jones. *Probabilistic Non-Determinism*. PhD thesis, Univ. of Edinburgh, 1990.

[25] R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2:95–112, 1996.

[26] Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer jan De Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175:93–125, 1997.

[27] Oleg Kiselyov and Chung-chieh Shan. Monolingual probabilistic programming using generalized coroutines. In *Uncertainty in Artificial Intelligence*, 2008.

[28] Achim Klenke. *Probability Theory: A Comprehensive Course*. Springer, 2006. ISBN 978-1-84800-047-6.

[29] Daphne Koller, David McAllester, and Avi Pfeffer. Effective Bayesian inference for stochastic programs. In *14th National Conference on Artificial Intelligence*, August 1997.

[30] Dexter Kozen. Semantics of probabilistic programs. In *Foundations of Computer Science*, 1979.

[31] Daniel Leivant. Higher order logic. In *In Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 229–321. Clarendon Press, 1994.

[32] Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science*, 2008.

[33] Sam Lindley, Philip Wadler, and Jeremy Yallop. The arrow calculus. *Journal of Functional Programming*, 20:51–69, 2010.

[34] David J. Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS – a Bayesian modelling framework. *Statistics and Computing*, 10(4), 2000.

[35] Robert Mateescu and Rina Dechter. Mixed deterministic and probabilistic networks. *Annals of Mathematics and Artificial Intelligence*, 2008.

[36] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), 2008.

[37] 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.

[38] James R. Munkres. *Topology*. Prentice Hall, second edition, 2000.

[39] Paul J. Nahin. *Duelling Idiots and Other Probability Puzzlers*. Princeton University Press, 2000.

[40] Russell O'Connor. Certified exact transcendental real number computation in Coq. In *TPHOLs'08*, pages 246–261, 2008.

[41] Toby Ord. The many forms of hypercomputation. *Applied Mathematics and Computation*, 178:143–153, 2006.

[42] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. *Transactions on Programming Languages and Systems*, 31(1), 2008.

[43] L. C. Paulson. Set theory for verification: II. Induction and recursion. *Journal of Automated Reasoning*, 15:167–215, 1995.

[44] Lawrence C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11:353–389, 1993.

[45] Avi Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In *Statistical Relational Learning*. MIT Press, 2007.

[46] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Principles of Programming Languages*, 2002.

[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.