3

An Application of Computable Distributions to the Semantics of Probabilistic Programs

Daniel Huang
University of California, Berkeley
Greg Morrisett
Cornell University
Bas Spitters
Aarhus University

Abstract: In this chapter, we explore how (Type-2) computable distributions can be used to give both (algorithmic) sampling and distributional semantics to probabilistic programs with continuous distributions. Towards this end, we sketch an encoding of computable distributions in a fragment of Haskell and show how topological domains can be used to model the resulting PCF-like language. We also examine the implications that a (Type-2) computable semantics has for implementing conditioning. We hope to draw out the connection between an approach based on (Type-2) computability and ordinary programming throughout the chapter as well as highlight the relation with constructive mathematics (via realizability).

3.1 Overview

Probabilistic programs exhibit a tension between the *continuous* and the *discrete*. On one hand, we are interested in using probabilistic programs to model natural phenomena—phenomena that are often modeled well with *reals* and *continuous distributions* (*e.g.*, as in physics and biology). On the other hand, we are also bound by the fundamentally *discrete nature of computation*, which limits how we can (1) represent models as programs and then (2) compute the results of *queries* on the model. The aim of this chapter[1 2 3] is to keep this tension in the fore by using the notion of a *(Type-2) computable distribution* as a lens through which to understand probabilistic programs. We organize our exploration via a series of questions.

[a] From *Foundations of Probabilistic Programming*, edited by Gilles Barthe, Joost-Pieter Katoen and Alexandra Silva published 2020 by Cambridge University Press.

[1] This chapter contains material from Huang (2017) and Huang and Morrisett (2016).

[2] Daniel Huang was supported by DARPA FA8750-17-2-0091.

[3] Bas Spitters was partially supported by the Guarded homotopy type theory project, funded by the Villum Foundation, project number 12386 and partially by the AFOSR project 'Homotopy Type Theory and Probabilistic Computation', 12595060. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the AFOSR.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

(i) *What is a (Type-2) computable distribution* (Section 3.2)? First, we review *Type-2 computability* (*e.g.*, see Weihrauch, 2000) and how it applies to reals and continuous distributions. The high-level idea is to represent continuum-sized objects as a sequence of discrete approximations that converge to the appropriate object instead of abstracting the representation of such an object.

(ii) *How do we implement continuous distributions as a library in a general-purpose programming language* (Section 3.3)? After we have seen the basic idea behind Type-2 computability, we sketch an implementation of reals and continuous distributions in a fragment of Haskell. We emphasize that the implementation does not assume any reals, continuous distributions, or operations on them as black-box primitives.

(iii) *What mathematical structures can we use to model such a library* (Section 3.4)? Our next step is to find mathematical structures that can be used to faithfully model the implementation. Towards this end, we review *topological domains* (*e.g.*, see Battenfeld et al., 2007; Battenfeld, 2008), which are an alternative to traditional structures used in denotational semantics. Topological domains support all the standard domain-theoretic constructions needed to model PCF-like[4] languages as well as capture the notion of (Type-2) computability. In particular, we can encode reals and continuous distributions as topological domains so that they are suitable for our purposes of giving semantics.

(iv) *What does a semantics for a core language look like* (Section 3.5)? In this section, we make the connection between the implementation and the mathematics more concrete by using the constructs described previously to give both (algorithmic) sampling and distributional semantics to a core PCF-like language extended with reals and continuous distributions (via a probability monad) called λ_{CD}.[5] The sampling semantics can be used to guide implementation while the distributional semantics can be used for equational reasoning.

(v) *What are the implications of taking a (Type-2) computable viewpoint for Bayesian inference* (Section 3.6)? Perhaps surprisingly, at least to those who employ Bayesian inference in practice, it can be shown that *conditioning* is not (Type-2) computable (see Ackerman et al., 2011). Hence, there is a sense in which a "Turing-complete" probabilistic programming language cannot support conditional queries for every expressible probabilistic model. Fortunately, we do not run into these pathologies in practice and can recover conditioning in sufficiently general settings.

We hope to draw out the connection between an approach based on (Type-2)

[4] Recall that the PCF (Programming Computable Functions) language is a core calculus that can be used to model typed functional languages such as Haskell.

[5] λ_{CD} also supports distributions on any countably based space. This means that λ_{CD} does not (in general) have distributions on function spaces, although the language itself contains higher-order functions.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

computability with ordinary programming (*i.e.*, programming in a fragment of Haskell) throughout the chapter as well as highlight the relation with constructive mathematics via *realizability* (*e.g.*, see Streicher, 2008) (Section 3.4.4).

Prerequisites We assume basic knowledge of programming language semantics (*e.g.*, at the level of Gunter, 1992). For our purposes, this primarily includes (1) the application of category theory to programming language semantics and (2) the use of complete partial orders (CPOs) to model the semantics of PCF. As we will be giving examples in Haskell, familiarity with the Haskell programming language will also be assumed.[6] Finally, we assume basic knowledge of measure-theoretic probability (*e.g.*, see Durrett, 2010).

3.2 Computability Revisited

What is a computable distribution? One approach to studying computability is based on Turing machines (*e.g.*, see Sipser, 2012). Under this approach, we define (1) a *machine model* (*i.e.*, the Turing machine) and (2) conditions under which the machine model is said to *compute*. More concretely, a Turing machine is said to *compute* a (partial) function $f : \Sigma^* \rightharpoonup \Sigma^*$ if it halts with $f(w) \in \Sigma^*$ on the output tape given $w \in \Sigma^*$ on an input tape, where Σ is a finite set and $\Sigma^* \triangleq \{a_0 \dots a_n \mid a_i \in \Sigma, 0 \leq i \leq n\}$ is a collection of words comprised of characters from Σ. The two element set $\mathbf{2} \triangleq \{0, 1\}$ for bits (or booleans) is a commonly used alphabet.

This definition of computability reveals that traditional computation is fundamentally *discrete*. We can see this directly in the definition of a computable function (with type $\Sigma^* \rightharpoonup \Sigma^*$), which maps elements of a discrete domain (*i.e.*, a set of finite words Σ^*) to elements of a discrete codomain (*i.e.*, a set of finite words Σ^* again). As Σ^* is countable, it cannot be put in bijection with the reals $\mathbb{R}$; hence, we cannot encode all the reals on a Turing machine.

One immediate issue that this highlights for probabilistic programs is how one should handle reals and continuous distributions while maintaining the connection back to computation. A pragmatic solution to this is to use floating point arithmetic, *i.e.*, discretize and finitize the reals. From this perspective, we can model the semantics of probabilistic programs using floating point numbers and finitely-supported discrete distributions (on floats) so that the semantics more faithfully models an actual implementation. Nevertheless, we sacrifice the correspondence between the program and the mathematics that we use on pencil-paper. An alternative to the situation above is to generalize the notion of computability to continuum-sized

[6] Familiarity with other typed functional languages such as ML should also suffice, although we should remind ourselves that Haskell has call-by-need semantics so that it has a lazy order-of-evaluation.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

sets in such a way that the computations can still by implemented by a standard machine.

3.2.1 Type-2 Computability

Type-Two Theory of Effectivity (abbreviated TTE, see Weihrauch, 2000) changes the conditions under which a machine is said to compute an answer but keeps the machine model as is. In this setting, a machine is said to *compute* a function $f : \Sigma^\omega \rightharpoonup \Sigma^\omega$ if it can write any initial segment of $f(w) \in \Sigma^\omega$ on the output tape in finite time given $w \in \Sigma^*$ on an input tape, where $\Sigma^\omega \triangleq \{a_0 a_1 \dots \mid a_i \in \Sigma, i \in \mathbb{N}\}$ is the set of streams composed of symbols from the finite set Σ. The set Σ^ω has continuum cardinality, and hence, can represent the reals and a class of distributions (Section 3.2.2). Once we *represent* continuum-sized objects on a machine, we have an avenue for studying which functions are Type-2 computable. Throughout the rest of the chapter, we will abbreviate Type-2 computable as computable[7] and use Type-2 computable for emphasis. We now review computable reals and distributions.

3.2.2 Computability, Reals and Distributions

Computability and reals Intuitively, we can represent a real on a machine by encoding its binary expansion. More formally, we represent a real $x \in \mathbb{R}$ on a machine by encoding a fast Cauchy sequence of rationals that converges to x. Recall that a sequence $(q_n)_{n\in\mathbb{N}}$ where each $q_n \in \mathbb{Q}$ is *Cauchy* if for every $\epsilon > 0$, there is an N such that $|q_n - q_m| < \epsilon$ for every $n, m > N$. Thus, the elements of a Cauchy sequence become closer and closer to one another as we traverse the sequence. When $|q_n - q_{n+1}| < 2^{-n}$ for all n, we call $(q_n)_{n\in\mathbb{N}}$ a *fast Cauchy sequence*. Hence, the representation of a real as a fast Cauchy sequence evokes the idea of enumerating its binary expansion. A real $x \in \mathbb{R}$ is *computable* if we can enumerate (uniformly in an enumeration of rationals) a fast Cauchy sequence that converges to x.

We give some examples of reals encoded as fast Cauchy sequences now.

Example 3.1 (Rational) Consider two encodings of 0 as a fast Cauchy sequence below.

(Constant) Let $(x_n)_{n\in\mathbb{N}}$ where $x_n \triangleq 0$ for $n \in \mathbb{N}$.
(Thrashing) Let $(y_n)_{n\in\mathbb{N}}$ where $y_n \triangleq \frac{1}{(-2)^{n+1}}$ for $n \in \mathbb{N}$.

As 0 itself is also a rational number, we can simply represent it as a constant 0 sequence given by $(x_n)_{n\in\mathbb{N}}$. We can also represent 0 as the sequence $(y_n)_{n\in\mathbb{N}}$, where

[7] Computability in the ordinary sense refers to Type-1 computability.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

the sequence jumps back and forth between positive and negative fractional powers of two as it converges towards 0. 0 is clearly a computable real.

Example 3.2 (Irrational) Let $x_n \triangleq 2 + \sum_{k=2}^{2+n} \frac{1}{k!}$. Then $(x_n)_{n\in\mathbb{N}}$ is a fast Cauchy encoding of e. It is easy to see that e is a computable real.

Example 3.3 (Non-computable) Every real can be expressed as a fast Cauchy sequence so there are necessarily non-computable reals as well. Let $(M_n)_{n\in\mathbb{N}}$ be some enumeration of Turing machines. Let $t_0 \triangleq 1$ and

$$t_{n+1} \triangleq \begin{cases} 2 \cdot t_n & M_n \text{ halts} \\ 1 + t_n & M_n \text{ does not halt.} \end{cases}$$

Then $(x_n)_{n\in\mathbb{N}}$ where $x_n \triangleq \sum_{i=0}^{n} \frac{1}{2^{t_i}}$ is a fast Cauchy sequence that is not computable because the Halting problem is not decidable.

A function $f : \mathbb{R} \to \mathbb{R}$ is *computable* if given a (fast Cauchy) sequence converging to $x \in \mathbb{R}$, there is an algorithm that outputs a (fast Cauchy) sequence converging to $f(x)$.[8] We emphasize that the algorithm must work generically for any input (fast Cauchy) sequence including the non-computable ones. we give some examples now.

Example 3.4 (Addition) The function $+_0 : \mathbb{R} \to \mathbb{R}$ that adds 0 is computable because an algorithm can obtain a (fast Cauchy) output sequence by adding the (fast Cauchy) input sequence element-wise to a (fast Cauchy) sequence of 0.

Most familiar functions are computable (*e.g.*, subtraction, multiplication, inverses, exponentiation, logarithms on non-negative reals, and trigonometric functions such as sines and cosines) so that there is an algorithm that transforms (fast Cauchy) inputs into (fast Cauchy) outputs. Nevertheless, there are familiar functions that are not computable.

Example 3.5 (Non-computable) Consider the function $=_0 : \mathbb{R} \to \mathbf{2}$ that tests if the input is equal to 0 or not. Intuitively, this function is not computable because we need to check the entire input sequence. For example, to check that the constant sequence is equivalent to the thrashing sequence, we have to check the entirety of both sequences, which cannot be done in finite time.

Computable metric spaces Topological spaces enable us to build a more general notion of computability on a space.[9] For the purposes of introducing reals and distributions, we consider topological spaces with a notion of distance, *i.e.*, *metric spaces*. As a reminder, a *metric space* (X, d) is a set X equipped with a *metric*

[8] A function $f : \mathbb{R}^n \to \mathbb{R}$ is computable if given (fast Cauchy) sequences converging to $x_1, \ldots, x_n \in \mathbb{R}$, there is an algorithm that outputs a (fast Cauchy) sequence converging to $f(x_1, \ldots, x_n)$.

[9] For more background on topology, we refer the reader to Munkres (2000).

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

$d : X \times X \to \mathbb{R}$. A metric induces a collection of sets called *(open) balls*, where a ball centered at $c \in X$ with radius $r \in R$ is the set of points within r of c, *i.e.*, $B(c,r) \triangleq \{x \in X \mid d(c,x) < r\}$. The topology $\mathcal{O}(X)$ associated with a metric space X is the one induced by the collection of balls. Hence, the open balls of a metric space provide a notion of distance in addition to providing a notion of approximation.

Example 3.6 $(\mathbb{N}, d_{\text{Discrete}})$ endows the naturals $\mathbb{N}$ with the discrete topology (*i.e.*, $\mathcal{O}(\mathbb{N}) = 2^{\mathbb{N}}$), where d_{Discrete} is the discrete metric (*i.e.*, $d(n,m) \triangleq 0$ if $n = m$ and $d(n,m) \triangleq 1$ otherwise for $n, m \in \mathbb{N}$).

Example 3.7 $(\mathbb{R}, d_{\text{Euclid}})$ endows the reals $\mathbb{R}$ with the familiar Euclidean topology, where d_{Euclid} is the standard Euclidean metric (*i.e.*, $d_{\text{Euclid}}(x,y) \triangleq |x - y|$).

Example 3.8 $(\mathbf{2}^\omega, d_{\text{Cantor}})$ endows the set of bit-streams $\mathbf{2}^\omega$ with the Cantor topology, where d_{Cantor} is defined as

$$d_{\text{Cantor}}(x,y) \triangleq \inf\{\frac{1}{2^n} \mid x_n \neq y_n\}.$$

One can check that a basic open set of the Cantor topology is of the form $a_1 \ldots a_n \mathbf{2}^\omega \triangleq \{b_1 b_2 \cdots \in \mathbf{2}^\omega \mid b_i = a_i, 1 \leq i \leq n\}$. That is, basic open sets of Cantor space fix finite-prefixes.

A computable metric space imposes additional conditions on a metric space so that a machine can enumerate successively more accurate approximations (according to the metric) of a point in the metric space. We need two additional definitions before we can state the definition. First, we say S is *dense* in X if for every $x \in X$, there is a sequence $(s_n)_{n\in\mathbb{N}}$ that converges to x, where $s_n \in S$ for every n. Second, we say that (X,d) is *complete* if every Cauchy sequence comprised of elements from X also converges to a point in X.

Definition 3.9 A *computable metric space* is a tuple (X,d,S) such that (1) (X,d) is a complete metric space, (2) S is a countable, enumerable, and dense subset, and (3) the real $d(s_i, s_j)$ is computable for $s_i, s_j \in S$ (see Hoyrup and Rojas, 2009, Def. 2.4.1).

Example 3.10 $(\mathbb{R}, d_{\text{Euclid}}, \mathbb{Q})$ is a computable metric space for the reals where we use the rationals $\mathbb{Q}$ as the approximating elements. Note that we can equivalently use dyadic rationals as the approximating elements instead of $\mathbb{Q}$.

Computability and distributions A distribution over the computable metric space (X,d,S) can be formulated as a point of the computable metric space

$$(\mathcal{M}(X), d_\rho, \mathcal{D}(S)),$$

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

where $\mathcal{M}(X)$ is the set of Borel probability measures on a computable metric space (X, d, S), d_ρ is the Prokhorov metric (see Hoyrup and Rojas, 2009, Defn. 4.1.1), and $\mathcal{D}(S)$ is the class of distributions with finite support at ideal points S and rational masses (see Hoyrup and Rojas, 2009, Prop. 4.1.1). The Prokhorov metric is defined as

$$d_\rho(\mu, \nu) \triangleq \inf\{\epsilon > 0 \mid \mu(A) \leq \nu(A^\epsilon) + \epsilon \text{ for every Borel } A\} ,$$

where $A^\epsilon \triangleq \{x \in X \mid d(x, y) < \epsilon \text{ for some } y \in A\}$. One can check that the sequence below converges (with respect to the Prokhorov metric) to the (standard) uniform distribution $\mathcal{U}(0, 1)$.

$$\left\{0 \mapsto \frac{1}{2}, \frac{1}{2} \mapsto \frac{1}{2}\right\}, \left\{0 \mapsto \frac{1}{4}, \frac{1}{4} \mapsto \frac{1}{4}, \frac{2}{4} \mapsto \frac{1}{4}, \frac{3}{4} \mapsto \frac{1}{4}\right\}, \ldots,$$

Thus, a uniform distribution can be seen as the limit of a sequence of increasingly finer discrete, uniform distributions. As with a computable real, we say that a distribution $\mu \in \mathcal{M}(X)$ is *computable* if we can enumerate (uniformly in an enumeration of a basis and rationals) a fast Cauchy sequence that converges to μ.

Although the idea of constructing a (computable) distribution as a (computable) point is fairly intuitive for the standard uniform distribution, it may be more difficult to perform the construction for more complicated distributions. Fortunately, we can also think of a distribution on a computable metric space (X, d, S) in terms of sampling, *i.e.*, as a Type-2 computable function $\mathbf{2}^\omega \rightharpoonup X$. To make this more concrete, we sketch an algorithm that samples from the standard uniform distribution given a stream of fair coin flips. The idea is to generate a value that can be queried for more precision instead of a sample x in its entirety.

Let $\mu_{\text{iid}}(a_1 \ldots a_n \mathbf{2}^\omega) \triangleq 1/2^n$ be the distribution associated with a stream of fair coin flips where 0 corresponds to heads and 1 corresponds to tails. A sampling algorithm will interleave flipping coins with outputting an element to the desired precision, such that the sequence of outputs $(s_n)_{n \in \mathbb{N}}$ converges to a sample. For instance, one binary digit of precision for a standard uniform distribution corresponds to obtaining the point $1/2$ because it is within $1/2$ of any point in the unit interval. Demanding another digit of precision produces either $1/4$ or $3/4$ according to the result of a fair coin flip. This is encoded below using the function `bisect`[10], which recursively bisects an interval n times, starting with $(0, 1)$, using the random bit-stream u to select which interval to recurse on.

$$\text{uniform} : (\texttt{Nat} \to \texttt{Bool}) \to (\texttt{Nat} \to \texttt{Rat})$$
$$\text{uniform} \triangleq \lambda u.\ \lambda n.\ \texttt{bisect}\ u\ 0\ 1\ n$$

In the limit, we obtain a single point corresponding to the sample.

[10] See the implementation of `stdUniform` in Section 3.3.2 for the full definition.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

The sampling view is (computably) equivalent to the view of a computable distribution as a point in an appropriate computable metric space. To state the equivalence, we need a few definitions. A *computable probability space* (X, μ) is a pair where X is a computable metric space and μ is a computable distribution (see Hoyrup and Rojas, 2009, Def. 5.0.1). We call a distribution μ on X *samplable* if there is a computable function $s : (\mathbf{2}^\omega, \mu_{\text{iid}}) \rightharpoonup (X, \mu)$ such that s is computable on dom(s) of full-measure (*i.e.*, $\mu(X) = 1$) and is measure-preserving (*i.e.*, $\mu = \mu_{\text{iid}} \circ s^{-1}$).

Proposition 3.11 (Computable iff samplable, see Freer and Roy, 2010, Lem. 2 and Lem. 3)**.** *A distribution* $\mu \in \mathcal{M}(X)$ *on computable metric space* $(X, d, \mathcal{S})$ *is computable iff it is samplable.*

Thus we can equivalently specify computable distributions by writing sampling algorithms.

3.3 A Library for Computable Distributions

How do we implement continuous distributions as a library in a general-purpose programming language? Our goal in this section is to translate the concepts about reals and distributions we saw previously in Section 3.2 into code. Towards this end, we sketch a Haskell library (Figure 3.1) that encodes reals and the sampling view of distributions.[11] We emphasize that the library does not assume any reals, continuous distributions, or operations on them as black-box primitives.

3.3.1 Library

The library consists of three modules. The first module `ApproxLib` provides the interface for computable metric spaces. The second module `RealLib` implements reals using the operations in `ApproxLib` and the third module `CompDistLib` implements (continuous) distributions. We go over the modules in turn now.

As we mentioned previously, the module `ApproxLib` provides abstractions for expressing elements as a sequence of approximations in a computable metric space. The core type exposed by the module is `Approx` τ, which models an element of a computable metric space and can be read as an approximation by a sequence of values of type τ. For example, a real can be given the type `Real` $\triangleq$ `Approx Rat`, meaning it is a sequence of rationals (`Rat`) that converges to a real. We form values of type `Approx` τ using `mkApprox` :: (`Nat` $\to \alpha$) $\to$ `Approx` α, which requires us to check[12] that the function we are coercing describes a fast Cauchy sequence, and project out approximations using `nthApprox` :: `Approx` $\alpha \to$ `Nat` $\to \alpha$.

[11] The code is available at https://github.com/danehuang/cdist-sketch.
[12] We do not use the Haskell type system to enforce that the function to coerce contains a fast Cauchy sequence so the caller of `mkApprox` needs to perform this check manually.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

```
module ApproxLib (Approx(..), CMetrizable(..), mkApprox,
    nthApprox) where

newtype Approx a = Approx { getApprox :: Nat -> a }

mkApprox :: (Nat -> a) -> Approx a  -- fast Cauchy sequence
nthApprox :: Approx a -> Nat -> a   -- project n-th approx.

class CMetrizable a where
    enum :: [a]                     -- countable, dense subset
    metric :: a -> a -> Approx Rat  -- computable metric


module CompDistLib (RandBits, Samp(..), mkSamp) where
import ApproxLib

type RandBits = Nat -> Bool
newtype Samp a = Samp { getSamp :: RandBits -> a }

mkSamp :: (CMetrizable a) => (RandBits -> Approx a) -> Samp (
    Approx a)
mkSamp = Samp

instance Monad Samp where
  ...                               -- see text
```

Figure 3.1 A Haskell library interface for expressing approximations in a computable metric space (module `ApproxLib`) and encoding (continuous) distributions (module `CompDistLib`). The library interface for reals (module `RealLib`) is not shown.

In order to form the type `Approx` τ, values of type τ should support the operations required of a computable metric space. We can indicate the required operations using Haskell's type-class mechanism.

```
class CMetrizable a where
    enum :: [a]
    metric :: a -> a -> Approx Rat
```

As a reminder, Haskell has lazy semantics so that the type `[`α`]` denotes a stream as opposed to a list. Thus `enum` corresponds to an enumeration of type α where α is the type of the dense subset. When we implement an instance of `CMetrizable` τ, we should check that the implementation of `enum` enumerates a dense subset and `metric` computes a metric as a computable metric space requires (see Section 3.2.2).

Below, we give an instance of `Approx Rat` for computable reals.

```
instance CMetrizable Rat where
    enum = 0 : [ toRational m / 2^n
             | n <- [1..]
```

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

```
                , m <- [-2^n * n..2^n * n]
                , odd m || abs m > 2^n * (n-1) ]
    metric x y = A (\_ -> abs (x - y))
```

This instance enumerates the dyadic rationals, which are a dense subset of the reals. Note that there are many other choices here for the dense enumeration.[13] In this instance, we can actually compute the metric as a dyadic rational, whereas a computable metric requires the weaker condition that we can compute the metric as a computable real.

Next, we can use the module `ApproxLib` to implement computable operations on commonly used types. For example, a library for computable reals will contain the `CMetrizable` τ instance implementation above and other computable functions. However, some operations are not realizable (*e.g.*, equality of reals) and so this module does not contain all operations one may want to perform on reals (*e.g.*, equality is defined on floats).

```
module RealLib (Real, pi, (+), ...) where
import ApproxLib

type Real = Approx Rat
instance CMetrizable Rat where
    ...

pi :: Real
(+) :: Real -> Real -> Real
-- etc.
```

The module `CompDistLib` contains the implementation of distributions. A sampler `Samp` α is a function from a bit-stream to values of type α.[14]

```
type RandBits = Nat -> Bool
newtype Samp a = Samp { getSamp :: RandBits -> a }
```

We can implement an instance of the sampling monad as below.

```
instance Monad Samp where
    return x = Samp (const x)
    (>>=) s f = Samp ((uncurry (getSamp . f)) . (pair (getSamp
        s . fst) snd) . split)
        where pair f g = \x -> (f x, g x)
              split = pair even odd
              even u = (\n -> u (2 * n))
              odd u = (\n -> u (2 * n + 1))
```

As expected, `return` corresponds to a constant sampler (`const`) that ignores its input randomness. The bind operator »= corresponds to a composition of samplers;

[13] Algorithms that operate on computable metric spaces compute by enumeration so the algorithm is sensitive to the choice of enumeration.

[14] The type `RandBits` is represented isomorphically as `Nat` $\to$ `Bool` instead of `[Bool]`.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

we first split (`split`) the input randomness into two independent streams (via `even` and `odd`), use one to sample from `s`, and continue with the other in `f`.

The module `CompDistLib` provides the function `mkSamp` to coerce an arbitrary Haskell function of the appropriate type into a value of type `Samp` α.

```
mkSamp :: (CMetrizable a) => (RandBits -> Approx a) -> Samp (
    Approx a)
mkSamp = Samp
```

We should call `mkSamp` only on sampling functions realizing Type-2 computable sampling algorithms.

3.3.2 Examples

We now encode discrete and continuous distributions using the constructs provided by library. These examples demonstrate how familiar distributions used in probabilistic modeling can be encoded in a Type-2 computable manner. As we walk through the examples, we will encounter some semantic issues that we would like a denotational semantics of probabilistic programs to handle. We will flag these in italics and revisit them after introducing a semantics for probabilistic programs (Section 3.5).

Discrete distribution Discrete distributions are much simpler compared to continuous distributions. Nevertheless, when paired with recursion, semantic issues do arise. For instance, consider the encoding of a geometric distribution with bias $1/2$, which returns the number of fair Bernoulli trials until a success. The distribution `stdBernoulli` denotes a Bernoulli distribution with bias $1/2$.

```
stdGeometric :: Samp Nat
stdGeometric = do
  b <- stdBernoulli
  if b then return 1
       else stdGeometric >>= return . (\n -> n + 1)
```

One possibility, although it occurs with zero probability, is for the draw from `stdBernoulli` to always be false. Consequently, `stdGeometric` diverges with probability zero. *A semantics should clarify the criterion for divergence and show that this recursive encoding actually denotes a geometric distribution.*

Continuous distributions Next, we fill in the sketch of the standard uniform distribution we presented earlier. As a reminder, we need to convert a random bit-stream into a sequence of (dyadic) rational approximations.

```
stdUniform :: Samp Real
stdUniform = mkSamp (\u -> mkApprox (\n -> bisect (n+1) u 0 1
    0))
```

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

```
    where
      bisect n u (l :: Rat) (r :: Rat) m
          | m < n && u m        =
              bisect n u l (midpt l r) (m+1)
          | m < n && not (u m) =
              bisect n u (midpt l r) r (m+1)
          | otherwise           =
              midpt l r
      midpt l r = l + (r - l) / 2
```

The function `bisect` repeatedly bisects an interval specified by (`l`, `r`). By construction, the sampler produces a sequence of dyadic rationals. We can see that this sampling function is uniformly distributed because it inverts the binary expansion specified by the uniformly distributed input bit-stream. Once we have the standard uniform distribution, we can encode other primitive distributions (*e.g.*, normal, exponential, etc.) as transformations of the uniform distribution as in standard statistics using return and bind.

For example, we give an encoding of the standard normal distribution using the Marsaglia polar transformation.

```
stdNormal :: Samp Real
stdNormal = do
  u1 <- uniform (-1) 1
  u2 <- uniform (-1) 1
  let s = u1 * u1 + u2 * u2
  if s < 1 then return (u1 * sqrt (log s / s))
         else stdNormal
```

The distribution `uniform` (-1) 1 is the uniform distribution on the interval $(-1,1)$ and can be encoded by shifting and scaling a draw from `stdUniform`. One subtle issue here concerns the semantics of $<$. As a reminder, equality on reals is not decidable. *Consequently, although we have used* $<$ *at the type* `Real` $\to$ `Real` $\to$ `Bool` *in the example, it cannot have the standard semantics of deciding between* $<$ *and* $\geq$.

Singular distribution Next, we give an encoding of the Cantor distribution. The Cantor distribution is singular so it is not a mixture of a discrete component and a component with a density. Perhaps surprisingly, this distribution is computable. The distribution can be defined recursively. It starts by trisecting the unit interval, and placing half the mass on the leftmost interval and the other half on the rightmost interval, leaving no mass for the middle, continuing in the same manner with each remaining interval that has positive probability. We can encode the Cantor distribution by directly transforming a random bit-stream into a sequence of approximations.

```
cantor :: Samp Real
cantor = mkSamp (\u -> mkApprox (\n -> go u 0 1 0 n))
```

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

```
  where
    go u (left :: Rat) (right :: Rat) n m
      | n < m && u n        =
        go u left (left + pow) (n + 1) m
      | n < m && not (u n) =
        go u (right - pow) right (n + 1) m
      | otherwise           =
        right - (1 / 2) * pow
      where pow = 3 ^^ (-n)
```

The sampling algorithm keeps track of which interval it is currently in specified by `left` and `right`. If the current bit is 1, we trisect the left interval. Otherwise, we trisect the rightmost interval. The number of trisections is bounded by the precision we would like to generate the sample to. *Crucially, the encoding makes use of the idea of generating a sample to arbitrary accuracy using a representation instead of the sample in its entirety.*

Partiality and distributions The next series of examples explores issues concerning distributions and partiality.

```
botSamp :: (CMetrizable a) => Samp (Approx a)
botSamp = botSamp

botSampBot :: (CMetrizable a) => Samp (Approx a)
botSampBot = mkSamp (\_ -> bot)
    where bot = bot
```

In the term `botSamp`, we define an infinite loop at the type of samplers. Intuitively, this corresponds to the case where we fail to provide a sampler, *i.e.*, an error in the worst possible way. In the term `botSampBot`, we produce a sampler that fails to generate a sample to any precision. In other words, we provide a sampler that is faulty in the worst possible way. We can try to observe the differences in the implementation (if any).

```
alwaysDiv :: Samp Real
alwaysDiv = do
  _ <- botSamp ::
         Samp Real
  stdUniform
```

```
neverDiv :: Samp Real
neverDiv = do
  _ <- botSampBot ::
         Samp Real
  stdUniform
```

If we run the term `alwaysDiv` on the left, we will see that the program always diverges. When we run the term `neverDiv` on the right, we will draw from the sampler `botSampBot` but discard the result. Due to Haskell's lazy semantics, this computation is ignored and the entire term behaves as a standard uniform distribution. *We would like a denotational semantics to reflect the differences in the operational behavior between these two terms.*

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

Commutativity and independence We end by considering the difference between a sampling and distributional interpretation of probabilistic programs. Below, we give equivalent encodings of distributions by commuting the order of sampling from independent distributions, but leaving everything else fixed.

```
myNormal :: Samp Real
myNormal = do
  x <- normal (-1) 1
  y <- normal (1) 1
  return (x + y)
```

```
myNormal' :: Samp Real
myNormal' = do
  y <- normal (1) 1
  x <- normal (-1) 1
  return (x + y)
```

From a sampling perspective, the two distributions are not strictly equivalent because the stream of random bits is consumed in a different order; consequently, the samples produced by `myNormal` and `myNormal'` may be different. *Thus, while a sampling semantics is easily implementable, we would also like a distributional semantics to enable reasoning about the distributional equivalence of programs. For instance, this would enable us to reason that two different sampling algorithms for the same distribution are equivalent.*

3.3.3 Notes

The implementation we have sketched is a proof of concept that shows that we can realize the interface by implementing computable distributions and operations on them as Haskell code. We note that there are multiple approaches to coding up Type-2 computability as a library. One prominent alternative is given by synthetic topology (Escardó, 2004), which assumes that the function space in the programming language used to code up topological results is continuous and derives the notion of an open set. These ideas can be used to help us structure an implementation.

One shortcoming of the library, and implementations of Type-2 computability more generally, is efficiency. We intend the presentation of the library as a means to sketch the connection of the computation with the mathematics. In practice, there are still reasons for using floating point arithmetic. First, inference algorithms are computationally intensive, even assuming operations on reals and distributions are constant-time, so one is willing to make tradeoffs for efficiency. Second, it is not necessary to compute answers to arbitrary accuracy for most applications. Notably, most inference algorithms already make approximations as the solutions to many interesting models are analytically intractable. Thus, there is still a (large) gap in practice between semantics and implementation. For ideas on how to implement Type-2 computability efficiently, we refer the reader to Bauer and Kavkler (2008) and Lambov (2007).

Lastly, in our description of the library, we have elided one important detail. One computable function we need to encode is the *modulus* of a computable function

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

between computable metric spaces. The modulus $g : (X \to Y) \to \mathbb{N} \to \mathbb{N}$ of a computable function $f : X \to Y$ between computable metric spaces $(X, d_X, \mathcal{S}_X)$ and $(Y, d_Y, \mathcal{S}_Y)$ is a function that computes the number of input approximations consumed to produce an output approximation to a specified precision. For example, if the algorithm realizing f looks at $s^X_{i_0}, \dots, s^X_{i_{41}}$ to compute an output $s^Y_{i_n}$ such that $d_Y(s^Y_{i_n}, f(x)) < 2^{-(n+1)}$ and $(s^X_{i_m})_{m \in \mathbb{N}} \to x$, then the modulus $g(f)(n)$ is 42. Within a machine model, one can simply "look at the tape and head location" to obtain the modulus. However, one can show that the modulus of continuity is not expressible in a functionally-extensional language. This in essence follows from the fact that the modulus of two extensionally equivalent functions may not be equivalent. We can use Haskell's imprecise exceptions mechanism (see Peyton Jones et al., 1999), an impure feature, in a restricted manner to express the modulus.[15]

3.4 Mathematical Structures for Modeling the Library

What mathematical structures can we use to model such a library? Now that we have seen that we can implement reals and continuous distributions in code, our next task is to find mathematical structures that can be used to faithfully model the implementation. In doing so, we will set ourselves up for giving *denotational semantics* to probabilistic programs under the additional constraint that the model takes computability into account (Section 3.5).

Towards this end, we review *topological domains*, an alternative to traditional domain theory (Section 3.4.1). Topological domains support all the standard domain-theoretic constructions needed to model PCF-like languages as well as capture the notion of Type-2 computability, and hence, can form the basis of a semantics for PCF-like languages. Next, we encode distributions as topological domains. We do this for a sampling view (Section 3.4.2) and a distributional view (Section 3.4.3) based on *valuations*, a topological variant of a measure. We also construct a probability monad (Giry, 1982) on countably based topological (pre)domains, which includes computable metric spaces, so we can model the monadic implementation of distributions in the library.

Finally, we put the approach proposed here, which emphasizes Type-2 computability, in perspective. We begin by exploring an alternative approach to capturing Type-2 computability via *realizability* (Section 3.4.4). Roughly speaking, we can view a constructive logic as a "programming language" that we can use to program computable distributions. We end by reviewing alternative structures that can be used to model the semantics of probabilistic programs (Section 3.4.5).

[15] See http://math.andrej.com/2006/03/27/sometimes-all-functions-are-continuous.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

3.4.1 Domains and Type-2 Computability

In this section, we review *topological domains*. Unlike a CPO, a topological domain in general does not carry the Scott topology, and hence, does not consider the partial order primary. Instead, topological domains start with the topology as primary and derive the order. For a complete treatment, we refer the reader to Battenfeld (2008) and the references within (*e.g.*, see Battenfeld, 2004; Battenfeld et al., 2006, 2007). Towards this end, we will follow the overview given by Battenfeld et al. (2007) to introduce the main ideas, which constructs topological domains in two steps: (1) connecting computability to topology and (2) relating topology to order. Most of this overview can be skimmed upon a first read, although the examples will be helpful. At the end, we will summarize the relevant structure that makes topological domains good candidates for modeling probabilistic programs. In Section 3.5, we will use this structure to give semantics to a core language.

Computability to topology Topological domain theory starts with the observation that topological spaces provide a good model of *datatypes*. In short, a point in a topological space corresponds to an inhabitant of a datatype and the open sets of the topology describe the observable properties of points. Consequently, one can test if an inhabitant of a datatype satisfies an observable property by performing a (potentially diverging) computation that tests if the point is contained in an open set. To make use of this observation, topological domain theory builds off of the Cartesian closed category of qcb_0 spaces[16] (*e.g.*, see Escardó et al., 2004), a subcategory of topological spaces that makes the connection between computation and topology precise. It is helpful to introduce a qcb_0 space by way of a *represented space* which starts with the idea of realizing computations on a machine model before adding back the topological structure.

Definition 3.12 A *represented space* (X, δ_X) is a pair of a set X with a partial surjective function $\delta_X : \mathbf{2}^\omega \rightharpoonup X$ called a *representation*.

We call $p \in \mathbf{2}^\omega$ a *name* of x when $\delta_X(p) = x$. Thus, a name encodes an element of the base set X as a bit-stream which in turn can be computed on by a Turing machine. A *realizer* for a function $f : (X, \delta_X) \to (Y, \delta_Y)$ is a (partial) function $F : \mathbf{2}^\omega \rightharpoonup \mathbf{2}^\omega$ such that $\delta_Y(F(p)) = f(\delta_X(p))$ for $p \in \mathrm{dom}(f \circ \delta_X)$. A function $f : X \to Y$ between represented spaces is called *computable* if it has a computable realizer. It is called *continuous* if it has a continuous realizer (with respect to the Cantor topology).[17] Unfolding the definition of continuity of a (partial) function $f : \mathbf{2}^\omega \rightharpoonup \mathbf{2}^\omega$ on Cantor space shows that it encodes a *finite prefix property*—this means that a machine can

[16] qcb_0 stands for a T_0 quotient of a countably based space.

[17] Note that a continuous function $f : X \to Y$ between represented spaces does not mean that $f : X \to Y$ is a topologically continuous function with respect to the final topologies induced by the respective representations.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

compute $f(p)$ to arbitrary precision after consuming a finite amount of bits of p in finite time when f is continuous.

In order to relate the machine-model view to a topology so we can define a qcb_0 space, we will need a notion of an *admissible representation*. A representation δ_X of X is *admissible* if for any other representation δ'_X of X, the identify function on X has a continuous realizer (Battenfeld et al., 2007, Defn. 3.10).

Definition 3.13 A *qcb_0 space* is a represented space (X, δ_X) with *admissible representation* δ_X.

The topology is the *quotient topology* (or *final topology*) induced by the representation δ_X. If X and Y are qcb_0 spaces, then the topologically continuous functions between them coincide with those that have continuous realizers (Battenfeld et al., 2007, Cor. 3.13), which gives the same characterization as an admissible represented space. We give two examples of qcb_0 spaces to illustrate the corresponding realizers and topologies.

Example 3.14 Define the set $\mathbb{S} \triangleq \{\bot, \top\}$ with representation $\delta_{\mathbb{S}}(\bot) \triangleq 00\ldots$ and $\delta_{\mathbb{S}}(\top) \triangleq p$ for $p \neq 00\ldots$. Then $(\mathbb{S}, \delta_{\mathbb{S}})$ is a qcb_0 space known as *Sierpinski* space. In particular, Sierpinski space encodes the notion of semi-decidability—a Turing machine semi-decides that a proposition holds (encoded as $\top$) only if it eventually outputs a non-zero bit.

Example 3.15 Let (X, d, S) be a computable metric space. Then $(X, \delta_{\text{Metric}})$ is a qcb_0 space with admissible representation δ_{metric} that uses fast Cauchy sequences as names. More concretely, $(\delta_{\mathbb{Q}}(w_n))_{n\in\mathbb{N}} \to \delta_{\text{metric}}(p)$ where $\delta(p) = \langle\, w_1, w_2, \ldots\, \rangle$. As a special case, $(\mathbb{R}, \delta_{\mathbb{R}})$ is a represented space, where $\delta_{\mathbb{R}}$ is a representation that uses fast Cauchy sequences of rationals as names.

Topology to order The next piece of structure topological domain theory imposes is the order-theoretic aspect. The idea is to use the standard interpretation of recursive functions as the least upper bound of an ascending chain of the approximate functions obtained by unfolding. Because topological domain theory takes the topology as primary and the order as secondary, this task requires some additional work.

Recall that we can convert a topological space into a preordered set via the *specialization preorder*, which orders $x \sqsubseteq y$ if every open set that contains x also contains y. We write S to convert a topological space into a preordered set. Intuitively, $x \sqsubseteq y$ if x contains less information than y. For a metric space, we can always find an open ball that separates two distinct points x and y (because the distance between two distinct points is positive). Hence, the specialization preorder of a metric space always gives the discrete order (*i.e.*, information ordering), and hence degenerately, a CPO.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

Definition 3.16 (Battenfeld et al., 2007, Defn. 5.1)**.** A qcb_0 space is called a *topological predomain* if every ascending chain $(x_i)_{i \in \mathbb{N}}$ (with respect to the specialization preorder $\sqsubseteq$) has an upper bound x such that $(x_i)_{i \in \mathbb{N}} \to x$ (with respect to its topology).

Thus, we see in the definition that a topological predomain (1) builds off of a qcb_0 space and (2) ensures that least upper bounds of increasing chains exist. The former condition provides the topology and theory of effectivity while the latter condition prepares us for modeling least fixed-points. The following provides a useful characterization of qcb_0 spaces that relates the topology back to the order.

Definition 3.17 (Battenfeld et al., 2007, Defn. 5.3)**.** A topological space $(X, \mathcal{O}(X))$ is a *monotone convergence space* if its specialization order is a CPO and every open is Scott open.

Proposition 3.18 (Battenfeld et al., 2007, Prop. 5.4)**.** *A qcb_0 space is a topological predomain iff it is a monotone convergence space.*

Hence, we see that the Scott topology is in general finer than the topology associated with a topological predomain.

Analogous to standard domain theory, a topological predomain is called a *topological domain* if it has least element, written $\perp$, under its specialization order (Battenfeld et al., 2007, Defn. 5.6).

Proposition 3.19 (Battenfeld et al., 2007, Thm. 5.7)**.** *Every continuous endofunction on a topological domain has a least fixed-point.*

We look at the relation between order and topology more closely through a series of examples below.

Example 3.20 Consider the *discrete* CPO $(\mathbb{N}, \sqsubseteq_{\text{discrete}})$ with *discrete ordering* $\sqsubseteq_{\text{discrete}}$, (*i.e.*, $n \sqsubseteq_{\text{discrete}} m$ if $n = m$). The Scott topology on this CPO gives the *discrete topology*, *i.e.*, $\mathcal{O}(\mathbb{N}) = \{\{n\} \mid n \in \mathbb{N}\}$. The specialization preorder applied to the resulting topology gives back the original CPO. Thus, we additionally see that the topological predomain coincides with the CPO.

Example 3.21 Consider the CPO $(\{[a, 1) \mid a \in \mathbb{R}\} \cup \{[0, 1]\}, \subseteq)$ with ordering given by set inclusion. The Scott topology on this CPO gives the *lower topology*, *i.e.*, $\mathcal{O}([0, 1]) = \{(a, 1] \mid a \in [0, 1)\} \cup \{[0, 1]\}$. Like the previous example, the specialization preorder applied to the resulting topology gives back the original CPO. Hence, the topological domain also coincides with the CPO.

In the two examples above, we saw instances where the order and topology coincide. In the next two examples, we will see cases where they differ, thus highlighting differences between CPOs and topological (pre)domains.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

Construction	$D \times E$	$D \Rightarrow E$	$D + E$	$D \otimes E$	$D \Rrightarrow E$	$D \oplus E$	$D_\perp$
TP	✓+	✓+	✓+				✓
TD	✓+	✓+		✓	✓	✓	✓
TD$_!$	✓+	✓		✓	✓+	✓+	✓

Figure 3.2 Summary of constructs on topological predomains (category **TP**), topological domains (category **TD**), and topological domains with strict morphisms (category **TD**$_!$). (Compare this figure with one for CPOs (Abramsky and Jung, 1994, pg. 46).) The symbol ✓ indicates that the category is closed under that construct and the symbol + additionally indicates that it corresponds to the appropriate categorical construct.

Example 3.22 The reals $\mathbb{R}$ with Euclidean topology is a metric space, and hence, the specialization preorder gives a discrete CPO $(\mathbb{R}, \sqsubseteq_{\text{discrete}})$. However, the Scott topology of the resulting discrete CPO is the discrete topology. Hence, the topologies do not coincide.

Example 3.23 The Scott continuous functions from $\mathbb{R}$ to $\mathbb{R}$ contain all functions. However, the space of functions between the topological predomains $\mathbb{R}$ and $\mathbb{R}$ contain just the continuous ones.

The last example concerns modeling divergence for reals.

Example 3.24 The *partial reals* $\tilde{\mathbb{R}}$ (*e.g.*, see Escardó, 1996) can be modeled as (closed) intervals $[l, u]$ ordered by reverse inclusion where l is a lower-real and a u is an upper-real. The subspace of the maximal elements yields the familiar Euclidean topology. Note that $\tilde{\mathbb{R}}_\perp \neq \mathbb{R}_\perp$.

Categorical structure We end by summarizing the categorical structure of topological domains (Figure 3.2) applicable to giving semantics to probabilistic programs.[18] In short, topological (pre)domains possess essentially the same categorical structure as their CPO counterparts. Hence, we will be able to give semantics to programming languages using topological domains in much the same way that we use CPOs.

The relevant categories include **TP** (topological predomains and continuous functions),[19] **TD** (topological domains and continuous functions),[20] and **TD**$_!$ (topological domains and strict continuous functions).[21] We will use the notation below for categorical constructions with the usual semantics.

[18] We include sums $(D + E)$ and coalesced sums $(D \oplus E)$ for completeness. Similar to a smash product, a coalesced sum $D \oplus E$ identifies the least element of D with the least element of E.

[19] **TP** is a full reflective exponential ideal of **QCB** (category with qcb_0 spaces as objects and continuous functions as morphisms) (Battenfeld et al., 2007, Thm. 5.5).

[20] **TD** is an exponential ideal of **QCB** and is closed under countable products in **QCB** (Battenfeld et al., 2007, Thm. 5.9).

[21] **TD**$_!$ (1) is countably complete (limits inherited from **QCB**), (2) has countable coproducts, and (3) $\oplus$ and $\Rrightarrow$ (with $\mathbb{S}$ as unit) provides symmetry monoidal closed structure on **TD**$_!$ (Battenfeld et al., 2007, Thm. 6.1, Thm. 6.2, Prop. 6.4).

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

(*Function*) We write $D \Rightarrow E$ for continuous functions ($D \Rrightarrow E$ for strict continuous functions); the corresponding operation includes $\mathrm{eval} : (D \Rightarrow E) \times D \Rightarrow E$, $\mathrm{uncurry} : (D \Rightarrow E \Rightarrow F) \Rightarrow (D \times E \Rightarrow F)$, and $\mathrm{curry} : (D \times E \Rightarrow F) \Rightarrow D \Rightarrow E \Rightarrow F$. We will subscript function space $\Rightarrow$ with the appropriate category when it is not clear from context which function space we are referring to, *e.g.*, $D \Rightarrow_{\mathbf{TD}} E$.

(*Product*) We write $D \times E$ for products ($D \otimes E$ for smash products);[22] the corresponding operations include first projection $\pi_1 : D \times E \Rightarrow D$, second projection $\pi_2 : D \times E \Rightarrow E$, and pairing $\langle \cdot, \cdot \rangle : (D \Rightarrow E) \times (D \Rightarrow F) \Rightarrow (D \Rightarrow E \times F)$.

(*Lift*) $D_\perp$ lifts a *(pre)domain*; the corresponding operations include lifting elements $\lfloor \cdot \rfloor : D \Rightarrow D_\perp$, lifting the domain of a function $\mathrm{lift}_D : (D \Rightarrow E_\perp) \Rightarrow (D_\perp \Rightarrow E_\perp)$, lifting the codomain of a function $\mathrm{lift}_C : (D \Rightarrow E) \Rightarrow (D \Rightarrow E_\perp)$, and unlifting elements $\lceil \cdot \rceil : D_\perp \Rightarrow D$ for D ($\lceil \lfloor d \rfloor \rceil = d$ and undefined otherwise). Given a morphism $f : D \Rightarrow E$, we write $f_\perp : D_\perp \Rightarrow E_\perp$ to refer to the morphism with lifted domain and codomain.

3.4.2 *Sampling*

As a reminder, the library implementation converts an input bit-stream into a sample in the desired space. Hence, we begin by encoding the sampling implementation of distributions from the library as a topological domain.

Define an (endo)functor $\boldsymbol{S}$ that sends a topological predomain D to a sampler on D and a morphism to one that composes with the underlying sampler. Then, the topological domain $\boldsymbol{S}(D)$ is a sampler producing values in the lifted topological domain $D_\perp$.

Proposition 3.25 *The functor* $\boldsymbol{S}$ *defined as*

$$\boldsymbol{S}(D : \boldsymbol{TP}) \triangleq \mathbf{2}^\omega \Rightarrow D_\perp$$
$$\boldsymbol{S}(f : D \Rightarrow E) \triangleq s \mapsto f_\perp \circ s\,,$$

is well defined, where $\mathbf{2}^\omega$ *is the topological predomain equipped with the Cantor topology.*

The least element is one that maps all bit-streams to $\perp$. Next, we define three operations on samplers. The first operation creates a sampler that ignores its input bit-randomness and always returns d:

$$\mathrm{det} : D \Rightarrow \boldsymbol{S}(D)$$
$$\mathrm{det}(d) \triangleq \mathrm{const}(\lfloor d \rfloor)$$

[22] A smash product $D \otimes E$ identifies the least element of D with the least element of E.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

where const : $D \Rightarrow (E \Rightarrow D)$ produces a constant function.

The second operation splits an input bit-stream u into the bit-streams indexed by the even indices u_e and the odd indices u_o:

$$\text{split} : \mathbf{2}^\omega \Rightarrow \mathbf{2}^\omega \times \mathbf{2}^\omega$$
$$\text{split}(u) \triangleq (u_e, u_o)\,.$$

Note that if u is a sequence of independent and identically distributed bits, then both u_e and u_o will be as well.

The third operation sequences two samplers:

$$\text{samp} : \boldsymbol{S}(D) \times (D \Rightarrow \boldsymbol{S}(E)) \Rightarrow \boldsymbol{S}(E)$$
$$\text{samp}(s, f) \triangleq \text{uncurry}(\text{lift}_D(f)) \circ \langle s \circ \pi_1, \pi_2 \rangle \circ \text{split}\,.$$

It splits the input bit-randomness and runs the sampler s on one of the bit-streams obtained by splitting to produce a value. That value is fed to f, which in turn produces a sampler that is run on the other bit-stream obtained by splitting.

3.4.3 Valuations and a Probability Monad

Our goal now is encode distributions as *valuations* in the framework of topological domains. Once we have done so, we can interpret distribution terms in the library as elements of the appropriate topological domain. Next, we define the probability monad, which will be restricted to countably based topological (pre)domains. Consequently, the probability monad in λ_{CD} will be restricted to distributions on countably based spaces, which includes commonly used spaces such as reals and products of countably based spaces (Section 3.5).

Valuations and measures A valuation shares many of the same properties as a measure, and hence, can be seen as a topological variation of distribution.

Definition 3.26 A *valuation* $\nu : \mathcal{O}(X) \to [0, 1]$ is a function that assigns to each open set of a topological space X a probability such that it is (1) strict ($\nu(\emptyset) = 0$), (2) monotone ($\nu(U) \le \nu(V)$ for $U \subseteq V$), and (3) modular ($\nu(U) + \nu(V) = \nu(U \cup V) + \nu(U \cap V)$ for every open U and V).

One key difference between valuations and measures is that valuations are not required to satisfy countable additivity. Indeed, countable additivity is perhaps one of the defining features of a measure. We can rectify this situation for valuations by restricting attention to the *ω-continuous valuations*. As a reminder, a valuation ν is called *ω-continuous* if $\nu(\bigcup_{n \in \mathbb{N}} V_n) = \sup_{n \in \mathbb{N}} \nu(V_n)$ for $(V_n)_{n \in \mathbb{N}}$ an increasing sequence of opens. Hence, the countable additivity of μ encodes the ω-continuous

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

property. Importantly, note that every Borel measure μ can be restricted to the lattice of opens, written $\mu|_{O(X)}$, resulting in an ω-continuous valuation. Every Borel measure μ on X can be restricted to an ω-continuous valuation $\mu|_{O(X)}$: $[O^{\subseteq}(X) \Rightarrow_{\mathbf{CPO}} [0,1]^{\uparrow}]$ (see Schröder, 2007, Sec. 3.1). Moreover, μ is uniquely determined by its restriction to the opens $\mu|_{O(X)}$.[23] In other words, we can identify distributions on topological spaces with ω-continuous valuations.

Encoding valuations The presence of topological and order-theoretic structure suggests two strategies for encoding valuations as topological domains. In the first approach, we would take a realizer point of view as every topological domain is also a qcb_0 space. Under this approach, we would (1) define an admissible representation of the space of opens $O(X)$, (2) define an admissible representation of the interval $[0,1]$, and (3) verify that a representation of a valuation $O(X) \to [0,1]$ using the canonical function space representation is admissible and properly encodes a valuation. In the second approach, we would take an order-theoretic point of view. Under this approach, we would (1) verify that the space of opens $O(X)$ is a topological domain, (2) verify that the interval $[0,1]$ is a topological domain, and (3) verify that the continuous functions $O(X) \Rightarrow [0,1]$ encodes a valuation correctly. In either strategy, a common thread is that we need to encode the opens $O(X)$ and the interval $[0,1]$. We start with the realizer perspective.

Let $todo(X, \mathbb{S})$ be the space of continuous functions between the represented spaces X and $\mathbb{S}$. Let $[0,1]_< \triangleq ([0,1], \delta_<)$ be the represented space with representation $\delta_<$ that represents $r \in [0,1]$ as all the rational lower bounds. Next, we define the opens $O(X)$ and the interval $[0,1]$ for the order-theoretic perspective. Let $O^{\subseteq}(X) \triangleq (O(X), \subseteq)$ be the lattice of opens (and hence a CPO) of a topological space X ordered by subset inclusion. Let $[0,1]^{\uparrow} \triangleq ([0,1], \leq)$ be the interval $[0,1]$ ordered by $\leq$. The next proposition shows that the realizer perspective and the order-theoretic perspective are equivalent.

Proposition 3.27
(i) $[0,1]_< \cong [0,1]^{\uparrow}$ *and*
(ii) $todo(X, \mathbb{S}) \cong O^{\subseteq}(X)$ *when X is an admissible represented space.*[24]

The next proposition shows that the realizer and order-theoretic views are equivalent under the additional assumption that the base topological space is countably based.

Proposition 3.28 *Let $(X, O(X))$ be a countably based topological space.*
(i) $[O^{\subseteq}(X) \Rightarrow_{\mathbf{CPO}} [0,1]^{\uparrow}] \cong todo(O(X), [0,1]_<)$ *and*

[23] Note that the ω-continuous condition encodes what it means for a function to be ω-Scott continuous, *i.e.*, an ω-CPO continuous function.
[24] The second item is due to Schröder (2007, Thm. 3.3).

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

(ii) $[\mathcal{O}^{\subseteq}(X) \Rightarrow_{\boldsymbol{CPO}} [0,1]^{\uparrow}] \cong [\mathcal{O}^{\subseteq}(X) \Rightarrow_{\boldsymbol{TD}} [0,1]^{\uparrow}]$.[25]

Proposition 3.28 gives three equivalent views of a valuation as (1) a CPO continuous function, (2) a continuous map between represented spaces, and (3) a continuous function between topological domains. View (2) indicates that there is an associated theory of effectivity on valuations. We will use this view to give semantics to probabilistic programs.

Integration Similar to how one can integrate a measurable function with respect to a measure, one can integrate a lower semi-continuous function with respect to a valuation. Let X be a represented space and $\mu \in \mathcal{M}_1(X)$ where $\mathcal{M}_1(X)$ is the collection of Borel measures on X that have total measure 1.

Proposition 3.29 *The integral of a lower semi-continuous function* $f \in todo(X, [0,1]_<)$ *with respect to a Borel measure* μ

$$\int : todo(X, [0,1]_<) \times \mathcal{M}_1(X) \to [0,1]_<$$

is lower semi-continuous (see Schröder, 2007, Prop. 3.6). In fact, it is even lower semi-computable (Schröder, 2007, Prop. 3.6) (Hoyrup and Rojas, 2009, Prop. 4.3.1).

The integral is defined in an analogous manner to the Lebesgue integral, *i.e.*, as the limit of step functions on opens instead of measurable sets. The integral possesses many of the same properties, including Fubini and monotone convergence.

Probability monad Finally, we combine the results about valuations and integration to define a probability monad. Let $\mathbf{TP}_\omega$ be the full subcategory of $\mathbf{TP}$ where the objects are countably based. Define the (endo)functor $\boldsymbol{P}$ on countably based topological predomains that sends an object D to the space of valuations on D and a morphism to one that computes the pushforward.

Proposition 3.30 *The functor* $\boldsymbol{P}$ *defined as*

$$\boldsymbol{P}(D : \boldsymbol{TP}_\omega) \triangleq \mathcal{O}^{\subseteq}(D) \Rightarrow [0,1]^{\uparrow}$$
$$\boldsymbol{P}(f : D \Rightarrow E) \triangleq \mu \mapsto \mu \circ f^{-1}$$

is well defined.

It is straightforward to check that $\boldsymbol{P}$ is a functor. We can construct a probability monad using the functor $\boldsymbol{P}$.

[25] The first item is due to Schröder (2007, Sec. 3.1, Thm 3.5, Cor. 3.5). For the second item, recall that every ω-continuous pointed CPO with its Scott topology coincides with a topological domain (Battenfeld et al., 2007). The least element is the valuation that maps every open set to 0.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

Proposition 3.31 *The triple* $(\boldsymbol{P}, \eta, \gg_\flat)$ *is a monad, where*

$$\eta(x)(U) \triangleq \mathbb{1}_U(x)$$

$$(\mu \gg_\flat f)(U) \triangleq \int f_U \, d\mu \text{ where } f_U(x) = f(x)(U).$$

It is largely straightforward to check that $(\boldsymbol{P}, \eta, \gg_\flat)$ is a monad.[26]

3.4.4 Realizability

Sections 3.4.1, 3.4.2, and 3.4.3 taken together provide enough structure for giving semantics to probabilistic programs with continuous distributions. Thus, the reader interested in seeing the semantics “in action" in a core language can skip ahead to Section 3.5.

In this section, we explore another approach to Type-2 computability based on realizability. The primary motivation for doing so is that we will obtain another perspective on computability (*i.e.*, in addition to the topological and order-theoretic ones) that highlights the connection with *constructive* mathematics. Intuitively, we have a constructive object if we can *realize* the object as a *program*. As another source of motivation, it is also possible to give semantics to programming languages directly using the realizability approach (*e.g.*, see Longley, 1995). Hence, we will gain another method of giving semantics in addition to the traditional order-theoretic one.

Under the realizability approach, we will approach Type-2 computability using an abstract machine model, *i.e.*, a *partial combinatory algebra* (PCA) as opposed to a concrete machine model (*i.e.*, a Turing machine). A PCA consists of an underlying set X and a partial application function $\cdot : X \times X \rightharpoonup X$ subject to certain laws that ensure *combinatorial completeness*, *i.e.*, that a PCA can simulate untyped lambda calculus. Hence, we can think of a PCA as an algebraic take on substitution. We obtain ordinary Type-1 computability by instantiating a PCA over the naturals $\mathbb{N}$; the partial application function of a PCA $\cdot : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N}$ can be defined to simulate the computation of partial recursive functions. By extension, we obtain a Type-2 machine by instantiating a PCA over *Baire space* $\mathbb{B} \triangleq \mathbb{N} \to \mathbb{N}$; the partial application function of a PCA $\cdot : \mathbb{B} \times \mathbb{B} \rightharpoonup \mathbb{B}$ can be defined to simulate the computation over streams of naturals. In the rest of this section, our goal is to unpack the (well-known) connection between computability and constructive mathematics via realizability, and to show that the base spaces and constructions that are useful for giving semantics to probabilistic programs with continuous distributions can be realized appropriately.

[26] In the case of bind, we can check that the identities involving integrals holds via standard arguments (*e.g.*, see Jones, 1989).

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

Overview The phrase we have in mind is: "Computability is the realizability interpretation of constructive mathematics" (Bauer, 2005). The high-level idea is to encode familiar mathematical objects in an appropriate logic and derive computability as a consequence of having a sound interpretation. Programming up mathematical spaces and their operations will then correspond to encoding the space and their operations in the logic.

(*Logic*) The logic for our setting is *elementary analysis* (*e.g.*, see Lietz, 2004, Sec. 1.3.3) called EL. EL extends an intuitionistic predicate logic with (1) Heyting arithmetic, (2) a sort for Baire space `Baire` for encoding continuum-sized objects, and (3) primitive-recursion and associated operators.

(*Semantics*) The semantics for this setting includes the category $\mathbf{Asm}(\mathcal{K}_2)$ of assemblies over Kleene's second algebra $\mathcal{K}_2$ (*i.e.*, a PCA over Baire space) and the full subcategory $\mathbf{Mod}(\mathcal{K}_2)$ of modest sets over $\mathcal{K}_2$. For more details on assemblies and modest sets, we refer the reader to the relevant literature (*e.g.*, see Streicher, 2008; Bauer, 2000a; Birkedal, 1999). For our purposes, it suffices to recall that a modest set can be identified with a represented space and that an assembly is a represented space with a *multi-representation*. Hence, modest sets model datatypes and assemblies model intuitionistic logic.

Because we take a constructive vantage point, we will need to check that the semantics induced by the relevant encodings of familiar mathematical objects in the logic coincides with the usual interpretation. For our purposes, this means checking that encodings of objects such as reals and distributions in EL produce the expected semantics. Towards this end, recall that we can associate a theory of effectivity with a space by defining it as a quotient of Baire space $\mathbb{B}/_\sim$ by a *partial equivalence relation* (PER) $\sim$. A quotient by a PER allows us to construct quotients and subsets of Baire space in one go. We recall the conditions required of the relation $\sim$ for the constructive encoding to coincide with the classical interpretation below.

Definition 3.32 (Lietz, 2004, Prop. 3.3.2)**.** We write $\sim^*$ if

(*RF conservative class*) antecedents of implications contained in $\sim$ are almost negative;[27]

(*partial equivalence relation*) $EL \vdash \texttt{sym}(\sim) \wedge \texttt{trans}(\sim)$ where $\texttt{sym}(\sim) \triangleq \forall \alpha\, \beta : \texttt{Baire}.\, \alpha \sim \beta \leftrightarrow \beta \sim \alpha$ and $\texttt{trans}(\sim) \triangleq \forall \alpha\, \beta\, \gamma : \texttt{Baire}.\, \alpha \sim \beta \rightarrow \beta \sim \gamma \rightarrow \alpha \sim \gamma$; and

(*stability*) $EL \vdash \forall x\, y : \texttt{Baire}.\, \neg\neg(x \sim y) \rightarrow x \sim y$.

[27] More formally, whenever $A \rightarrow B$ is a subformula of $\sim$, then the antecedant A is almost negative. As a reminder, a formula is *almost negative* if it only contains existential quantifiers in front of prime (*i.e.*, atomic) formulas.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

Now we recall a sufficient condition for the constructive interpretation to coincide with the classical interpretation.

Proposition 3.33 (Lietz, 2004, Prop. 3.3.2)**.** *If* $\sim^*$*, then the interpretations of* ${}^{\mathbb{B}}/_{\sim}$ *in the categories* $\boldsymbol{Asm}(\mathcal{K}_2)$ *and* $\boldsymbol{Asm}_t(\mathcal{K}_2)$ *(i.e., the truth or classical interpretation) yield computably equivalent realizability structures.*

Encodings Before proceeding to the encodings of the sets of interest in EL, we define two enumerations that will be useful for constructing the encodings. Let $\pi_1\langle n,m\rangle = n$ and $\pi_2\langle n,m\rangle = m$ so that they are pairing functions on naturals (e.g., Cantor pairing function). We also overload the notation $\langle\alpha,\beta\rangle$ to pair $\alpha \in \mathbb{B}$ and $\beta \in \mathbb{B}$.

(*Integers*) Encode the integers as

$$\mathbb{Z} = {}^{\mathbb{N}\times\mathbb{N}}/_{=_{\mathbb{N}}}$$

where $\langle a,b\rangle =_{\mathbb{N}} \langle c,d\rangle$ if $a - d = c - b$ (*e.g.*, as in Bauer, 2000a, Sec. 5.5.1). In words, we can think of an integer as a difference of two naturals. We write `Int` to refer to the enumeration on $\mathbb{N}\times\mathbb{N}$.

(*Rationals*) Encode the rationals as

$$\mathbb{Q} = {}^{\mathbb{Z}\times(\mathbb{N}\backslash\{0\})}/_{=_{\mathbb{Q}}}$$

where $\langle p,q\rangle =_{\mathbb{Q}} \langle s,t\rangle$ if $p\cdot t = s\cdot q$ (*e.g.*, as in Bauer, 2000a, Sec. 5.5.1). In words, we can think of a rational as a ratio of an integer and a non-negative natural. We write `Rat` to refer to the enumeration on $\mathbb{Z}\times(\mathbb{N}\backslash\{0\})$. We write $\leq_{\mathbb{Q}}$ and $<_{\mathbb{Q}}$ to implement $\leq$ and $<$ respectively on rationals.[28]

(*Non-negative rationals*) Encode the non-negative rationals similarly to the rationals, where we replace $\mathbb{Z}$ with $\mathbb{N}$. We write `NonNegRat` to refer to the enumeration on $\mathbb{N}\times(\mathbb{N}\backslash\{0\})$. We write $<_{\mathbb{Q}^+}$ to implement $<$ on the non-negative rationals.

We now encode the base spaces as quotients of Baire space. In defining the quotient $\sim$, it is helpful to recall the encoding of the space first. For example, a *lower real* is an encoding of a real that enumerates all of its rational lower bounds. Hence, two lower reals will be related if their encodings enumerate the same lower bounds. As another example, we can encode reals as a fast Cauchy sequences. Hence, two reals will be related if their fast Cauchy sequences are suitably close to one another. We summarize useful quotient encodings of base spaces below.

Proposition 3.34

(Sierpinski) *Let* $\alpha \sim_{\mathbb{S}} \beta$ *if* $(\forall n : \texttt{Nat}.\, \alpha\, n = 0) \leftrightarrow (\forall n : \texttt{Nat}.\, \beta\, n = 0)$.

[28] Note that we have that $\langle p,q\rangle < \langle s,t\rangle$ if $p\cdot t < s\cdot q$ (*e.g.*, as in Bauer, 2000a, Sec. 5.5.1).

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

(Lower real) *Let* $\alpha \sim_{\mathbb{R}_<} \beta$ *if* $\forall q : \texttt{Rat}. (\forall n : \texttt{Nat}. q <_{\mathbb{Q}} \alpha\, n) \leftrightarrow (\forall n : \texttt{Nat}. q <_{\mathbb{Q}} \beta\, n)$.

(Lower non-negative real) *Let* $\alpha \sim_{\mathbb{R}^+_<} \beta$ *if* $\forall q : \texttt{NonNegRat}. (\forall n : \texttt{Nat}. q <_{\mathbb{Q}^+} \alpha\, n) \leftrightarrow (\forall n : \texttt{Nat}. q <_{\mathbb{Q}^+} \beta\, n)$.

(Upper real) *Let* $\alpha \sim_{\mathbb{R}_>} \beta$ *if* $\forall q : \texttt{Rat}. (\forall n : \texttt{Nat}. \alpha\, n <_{\mathbb{Q}} q) \leftrightarrow (\forall n : \texttt{Nat}. \beta\, n <_{\mathbb{Q}} q)$.

(Lifted partial real) *Let* $\langle \alpha_l, \alpha_u, \alpha_<, \alpha_> \rangle \sim_{\tilde{\mathbb{R}}} \langle \beta_l, \beta_u, \beta_<, \beta_> \rangle$ *if* $\alpha_l \sim_{\mathbb{R}_<} \beta_l \wedge \alpha_u \sim_{\mathbb{R}_>} \beta_u \wedge \alpha_< \sim_{\mathbb{S}} \beta_< \wedge \alpha_> \sim_{\mathbb{S}} \beta_>$.

(Real) *Let* $\alpha \sim_{\mathbb{R}} \beta$ *if* $\forall n : \texttt{Nat}. |\alpha\, n - \beta\, n| \leq_{\mathbb{Q}} 2^{-n+2}$.

We have $\sim^*_{\mathbb{S}}$, $\sim^*_{\mathbb{R}_<}$, $\sim^*_{\mathbb{R}^+_<}$, $\sim^*_{\mathbb{R}_>}$, $\sim^*_{\tilde{\mathbb{R}}}$, *and* $\sim^*_{\mathbb{R}}$.

It is largely straightforward to check that $\sim^*$ holds for the $\sim$ defined above.[29] Next, we state that semantic constructs can be encoded as quotients of Baire space as well.

Proposition 3.35 *Suppose* $\sim^*_X$ *and* $\sim^*_Y$.

(Lift) *Let* $\langle \alpha_C, \alpha_X \rangle \sim_{\perp} \langle \beta_C, \beta_X \rangle$ *if* $\alpha_C \sim_{\mathbb{S}} \beta_C \wedge \alpha_X \sim_X \beta_X$.

(Product) *Let* $\langle \alpha_X, \alpha_Y \rangle \sim_{X \times Y} \langle \beta_X, \beta_Y \rangle$ *if* $\alpha_X \sim_X \beta_X \wedge \alpha_Y \sim_Y \beta_Y$.

(Function) *Let* $\alpha \sim_{X \to Y} \beta$ *if* $\forall \gamma : \texttt{Baire}$, $\alpha \mid \gamma \sim_Y \beta \mid \gamma$ *where* $\alpha \mid \gamma$ *applies* α *to* γ *(in* $\mathcal{K}_2$*).*

We have $\sim^*_{\perp}$, $\sim^*_{X \times Y}$, *and* $\sim^*_{X \to Y}$.

It is straightforward to check that $\sim^*$ for the $\sim$ defined above.

We end by encoding valuations as quotients of Baire space. First, we need an enumeration of the open sets of a topological space. For a topological space $(X, \mathcal{O}(X))$, we can encode the collection of open sets as the function space $X \to \mathbb{S}$. As the measure of an open set is lower-semi computable (Proposition 3.4.3), a valuation can be encoded as an enumeration of pairs of a basic open and a non-negative lower real. For a countably based topological space with basis $\mathcal{B}(X)$, we have $\mathcal{B}(X) \cong \mathbb{N}$; hence, we can code a valuation as a sequence of non-negative lower reals.

Proposition 3.36 *Let* $\langle \alpha_1, \alpha_2, \dots \rangle \sim_{\mathcal{V}(X)} \langle \beta_1, \beta_2, \dots \rangle$ *if* $\forall n : \texttt{Nat}. \alpha_n \sim_{\mathbb{R}^+_<} \beta_n$. *Then* $\sim^*_{\mathcal{V}(X)}$.

Summary In summary, one view of what we have just seen is that we can use EL as a "programming language" (*i.e.*, a constructive logic as opposed to Haskell) for coding up mathematical structures relevant for probabilistic programs that have a notion of effectivity associated with them. In particular, the witnesses in the semantics of EL are given by elements of a PCA and modest sets over $\mathcal{K}_2$ can be identified with represented spaces (see Battenfeld et al., 2007, Sec. 8).

[29] For Sierpinski, see Lietz (2004, Defn. 3.2.4). For reals, see Bauer (2000b, Sec. 5.5.2). It is also useful to recall the notion of a *negative formula* (Bauer, 2000b, pg. 92) for checking the stability of $\sim$.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

3.4.5 Alternative Approaches

Probabilistic programs have a long history, and indeed, many structures have been proposed for modeling their semantics. Naturally, the choice of mathematical structure affects the language features that we can model. We close this section by reviewing a few of these alternative approaches as a point of comparison to the perspective given here that emphasizes Type-2 computability. We will focus on denotational approaches. There are also operational approaches to modeling the semantics of probabilistic programs (*e.g.*, see Park et al., 2005; Dal Lago and Zorzi, 2012).

One natural idea is to extend semantics based on CPOs to the probabilistic setting by putting distributions on CPOs. Saheb-Djahromi (1978) develops a probabilistic version of LCF by considering distributions on CPOs corresponding to base types (*i.e.*, booleans and naturals). Saheb-Djahromi also gives operational semantics as a Markov chain (described as a transition matrix) and shows that the operational semantics is equivalent to the denotational semantics. Jones (1989), in her seminal work, develops the theory of valuations on CPOs to further the study of distributions on CPOs via a *probabilistic powerdomain* $\mathcal{P}$. The probabilistic powerdomain is not closed under the function space; consequently, Jones interprets the function space $D \Rightarrow E$ probabilistically as $D \Rightarrow \mathcal{P}(E)$ (not $\mathcal{P}(D) \Rightarrow P(E)$).

Instead of taking order-theoretic structure as primary and extending it with probabilistic concepts, another idea is to take the probabilistic structure as primary and derive structure that models programming language constructs (*e.g.*, order-theoretic structure to model recursion). Kozen (1981) takes a structure amenable for modeling probability as primary (*i.e.*, Banach spaces) and imposes order-theoretic structure. This approach supports standard continuous distributions, although it does not support higher-order functions. In addition to the distributional semantics, Kozen also gives a sampling semantics and shows it equivalent to the distributional semantics. Danos and Ehrhard (2011) identify the category of probabilistic coherence spaces (PCSs) and use it to give denotational semantics to a probabilistic variant of PCF extended with (countable) choice. Hence, their approach supports discrete distributions. Ehrhard et al. (2014) show that PCSs provide a fully abstract model for probabilistic PCF so that the connection between the operational and denotational semantics is tight. Ehrhard et al. (2018) identify a Cartesian closed category of measurable cones and stable, measurable maps that is also order complete. They also provide an operational sampling semantics and show an adequacy result to link the denotational with operational semantics. This category can be used to model higher-order probabilistic languages with continuous distributions and recursion. Crubillé (2018) shows that the category of PCSs embeds into the (Cartesian closed) category of measurable cones with stable, measurable maps.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

One can also use measure-theoretic structure directly, although the category of measurable spaces with measurable maps is not Cartesian closed so higher-order functions cannot be modeled. Panangaden (1999) identifies a category of stochastic relations and shows how to use it to give denotational semantics to Kozen's first-order while language. The category has measurable spaces as objects and probability kernels as morphisms. Panangaden identifies (partially) additive structure in this category and uses it to interpret fix-points for Kozen's while language. Borgström et al. (2011) also interpret a type as a measurable space and use it to give denotational semantics to a first-order language without recursion based on measure transformers. They also show how to compile this language into a factor graph, which supports inference as well as provides an operational semantics. Staton (2017) shows how the category of measurable spaces with s-finite kernels can be used to give commutative semantics to a first-order language.

Another interesting approach considers alternatives to a measure-theoretic treatment of probability, but still considers the probabilistic structure as primary. Heunen et al. (2017) develop the theory of quasi-Borel spaces, which importantly, form a Cartesian closed category and show how quasi-Borel spaces can be used to model a higher-order probabilistic language with continuous distributions but without recursion. Vákár et al. (2019) show how to extend quasi-Borel spaces with order-theoretic structure so they can be used to model languages with recursion.

3.5 A Semantics for a Core Language

What does a semantics for a core language look like? Our goal in this section is to use the mathematical structures (*i.e.*, topological domains) we reviewed in the previous section to model a PCF-like language extended with reals and continuous distributions (via a probability monad) called λ_{CD}. We begin by introducing the syntax and statics of λ_{CD} (Section 3.5.1). As we might expect, the language features that we can model are restricted to the structure of the relevant topological domains. For instance, as we only define a probability monad on countably based spaces, the probability monad in λ_{CD} will be restricted to supporting only distributions on countably based spaces. This includes distributions on reals and products of countably based spaces, but does not include function spaces (although the language itself contains higher-order functions). Next, we give both (algorithmic) sampling and distributional semantics to λ_{CD} (Section 3.5.2). This illustrates more concretely the connection between the semantics and the library implementation of computable distributions. The structure of the semantics follows the usual one for PCF. Finally, we can use the core language and its semantics to resolve the semantic issues we raised when we sketched a library for computable distributions (Section 3.5.3).

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

$$
\begin{array}{rll}
\tau ::= & \texttt{Nat} \mid \tau \to \tau \mid \tau \times \tau \mid \texttt{Real} \mid \texttt{Dist}\ \tau & \\
M ::= & \texttt{0} \mid \texttt{succ} \mid \texttt{pred} \mid \texttt{if0}\ M\ M\ M & \text{(PCF-1)} \\
 & \mid x \mid \lambda x : \tau.\ M \mid M\ N \mid \texttt{fix}\ M & \text{(PCF-2)} \\
 & \mid (M, M) \mid \texttt{fst}\ M \mid \texttt{snd}\ M & \text{(products)} \\
 & \mid r \mid \mathit{rop} & \text{(reals)} \\
 & \mid \mathit{dist} \mid \texttt{return}\ M \mid x \leftarrow M\ ;\ M & \text{(distributions)}
\end{array}
$$

Figure 3.3 λ_{CD} extends a PCF-like language with products, reals, and distributions using a probability monad. The constructs for reals and distributions are shaded.

3.5.1 Syntax and Statics

Syntax The language λ_{CD} extends a PCF-like language with reals and distributions (Figure 3.3). The terms on lines *PCF-1* and *PCF-2* are standard PCF terms. The terms on the line marked *products* extend PCF with the usual constructions for pairs; (M, N) forms a pair of terms M and N, $\texttt{fst}\ M$ takes the first projection of the pair M, and $\texttt{snd}\ M$ takes the second projection of the pair M. The terms on the line marked *reals* add syntax for (1) constant reals r and (2) the application of primitive real functions *rop*. The terms on the line marked *distributions* add syntax for (1) primitive distributions *dist* and (2) return $\texttt{return}\ M$ and bind $x \leftarrow M\ ;\ N$ for an appropriate probability monad.

Statics Like PCF, λ_{CD} is a typed language. In addition to PCF types (*i.e.*, $\texttt{Nat}$ and $\tau \to \tau$), λ_{CD} includes the type of products ($\tau \times \tau$), reals ($\texttt{Real}$), and distributions ($\texttt{Dist}\ \tau$). Figure 3.4 summarizes the type-system for λ_{CD}. The expression typing judgement $\Gamma \vdash M : \tau$ is parameterized by a context Ψ (omitted in the rules) that contains the types of primitive distributions and functions.[30] The typing rules for the fragments marked *PCF-1*, *PCF-2*, and *products* is standard. The typing rules for the fragments marked *reals* and *distributions* are not surprising; nevertheless, we go over them as the constructs are less standard.

As expected, constant reals r are assigned the type $\texttt{Real}$. Primitive operations on reals *rop* (for real operation) have the type $\texttt{Real}^n \to \texttt{Real}$ where $\texttt{Real}^n \triangleq \texttt{Real} \times \cdots \times \texttt{Real}$ (n-times).

For expressions that operate on distributions, the judgement $\vdash_D \tau$ additionally enforces that the involved types are well formed. The distribution type $\texttt{Dist}\ \tau$ is well formed if the space denoted by τ is a computable metric space (Definition 3.9). This

[30] The full expression typing judgement would be written $\Psi; \Gamma \vdash M : \tau$. We omit Ψ because it is constant across typing rules.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

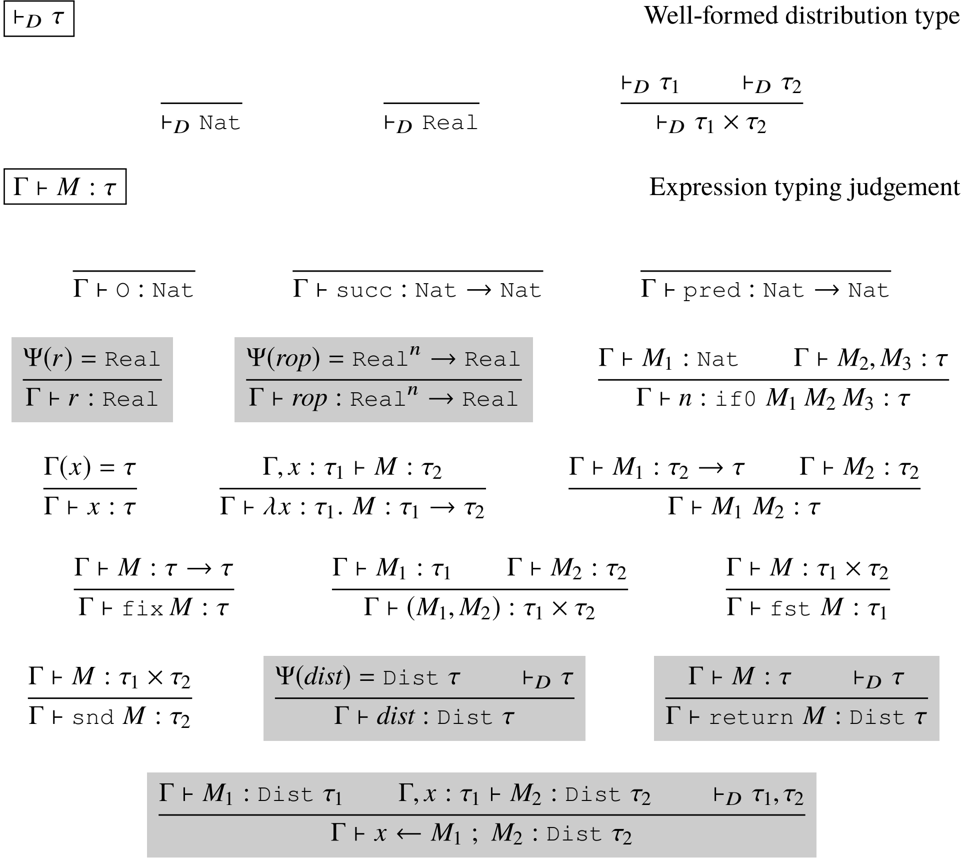

Figure 3.4 The type-system for λ_{CD}. The judgement $\vdash_D \tau$ checks that distribution types are well formed. The judgement $\Gamma \vdash M : \tau$ checks that expressions are well typed. The judgement is parameterized by a context Ψ (omitted), which contains that types of primitive distributions and functions. The typing rules for reals and distributions are shaded.

includes the type `Nat`, the type `Real` (Example 3.10), and products of well-formed types $\tau_1 \times \tau_2$.[31]

Given a term M that has a well-formed type, the construct `return` M corresponds to return in a probability monad and returns a point-mass centered at M. The typing rule for $x \leftarrow M \ ; \ N$ is the usual one for bind in a probability monad. The rule first checks that M has type `Dist` τ_1 and that τ_1 is well formed. Next, the rule checks that N under a typing context extended with $x : \tau_1$ has type `Dist` τ_2 and that τ_2 is well formed. The result is an expression of type `Dist` τ_2.

[31] As a reminder, we can also support distributions on any countably-based space (*e.g.*, distributions on distributions), but restrict our attention to these types for simplicity.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

$$\begin{aligned}
\mathcal{V}[\![\texttt{Nat}]\!] &\triangleq \mathbb{N}_\perp \\
\mathcal{V}[\![\tau_1 \to \tau_2]\!] &\triangleq (\mathcal{V}[\![\tau_1]\!] \Rightarrow \mathcal{V}[\![\tau_2]\!])_\perp \\
\mathcal{V}[\![\tau_1 \times \tau_2]\!] &\triangleq (\mathcal{V}[\![\tau_1]\!] \times \mathcal{V}[\![\tau_2]\!])_\perp \\
\mathcal{V}[\![\texttt{Real}]\!] &\triangleq \tilde{\mathbb{R}}_\perp \\
\mathcal{V}[\![\texttt{Dist}\ \tau]\!] &\triangleq \{(s, \mathrm{psh}_{\mathcal{V}[\![\tau]\!]}(s)) \mid s \in \boldsymbol{S}(\mathcal{V}[\![\tau]\!])\}
\end{aligned}$$

Figure 3.5 The interpretation of types $\mathcal{V}[\![\cdot]\!]$ denotes types as topological domains. We have shaded the interpretation of reals and distributions. Note that we are using a call-by-name interpretation.

3.5.2 Semantics

Interpretation of types The interpretation of types $\mathcal{V}[\![\tau]\!] \in \mathbf{TD}$ interprets a type τ as a topological domain and is defined by induction on types (Figure 3.5). The interpretation of types is similar to what one obtains from a standard CPO call-by-name interpretation.

For example, the interpretation of `Nat` lifts the topological domain $\mathbb{N}$. This is similar to the CPO interpretation of naturals as the lifted naturals. The interpretation of functions and products are the usual call-by-name interpretations, the difference being that we use the topological domain counterparts instead. The interpretation of the type of reals `Real` is a lifted partial real $\tilde{\mathbb{R}}_\perp$ (recall Example 3.24). The interpretation of the type of distributions `Dist` τ is a pair of a sampler and a distribution such that the sampler realizes the distribution. The (continuous) function $\mathrm{psh}_D : \boldsymbol{S}(D) \Rightarrow \boldsymbol{P}(D)$ computes the pushforward[32] and converts a sampler into its corresponding valuation. The well-formed distribution judgement $\vdash_D \tau$ ensures that the probability monad $\boldsymbol{P}$ is applied to only the countably based topological domains.

Denotation function The expression denotation function $\mathcal{E}[\![\Gamma \vdash M : \tau]\!] : \mathcal{V}[\![\Gamma]\!] \Rightarrow \mathcal{V}[\![\tau]\!]$ (see Proposition 3.38) is defined by induction on the typing derivation and is summarized in Figure 3.6. It is parameterized by a global environment Υ that interprets constant reals r, primitive functions *rop*, and primitive distributions *dist*. The global environment Υ should be well formed (defined shortly) with respect to the global context Ψ used in the expression typing judgement. After we introduce the notion of a well-formed global environment, we walk though the semantics and connect it with the library implementation, with a particular focus on the relation between a sampling and distributional view of probabilistic programs.

[32] We have that $\mathrm{psh}_D(s) \triangleq U \mapsto \int \mathbb{1}_U(\cdot)\, d\mu_s$ where $\mu_s \triangleq \mu_{\mathrm{iid}} \circ s^{-1}$.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

$$
\begin{aligned}
\mathcal{E}[\![\Gamma \vdash x : \tau]\!] &\triangleq \pi_x \\
\mathcal{E}[\![\Gamma \vdash \texttt{zero} : \texttt{Nat}]\!] &\triangleq \text{lift}_C \circ \text{const} \circ \Upsilon(\texttt{zero}) \\
\mathcal{E}[\![\Gamma \vdash \texttt{succ} : \texttt{Nat} \rightarrow \texttt{Nat}]\!] &\triangleq \text{lift}_C \circ \text{const} \circ \Upsilon(\texttt{succ}) \\
\mathcal{E}[\![\Gamma \vdash \texttt{pred} : \texttt{Nat} \rightarrow \texttt{Nat}]\!] &\triangleq \text{lift}_C \circ \text{const} \circ \Upsilon(\texttt{pred}) \\
\mathcal{E}[\![\Gamma \vdash \lambda x : \tau_1.\ M : \tau_1 \rightarrow \tau_2]\!] &\triangleq \text{lift}_C \circ \text{curry}(\mathcal{E}[\![\Gamma, x : \tau_1 \vdash M : \tau_2]\!]) \\
\mathcal{E}[\![\Gamma \vdash M_1\ M_2 : \tau_2]\!] &\triangleq \text{eval} \circ \langle \text{unlift}(\mathcal{E}[\![\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2]\!]), \mathcal{E}[\![\Gamma \vdash M_2 : \tau_1]\!] \rangle \\
\mathcal{E}[\![\Gamma \vdash \texttt{if0}\ M_1\ M_2\ M_3 : \tau]\!] &\triangleq \text{if0} \circ \langle \mathcal{E}[\![\Gamma \vdash M_1 : \texttt{Nat}]\!], \mathcal{E}[\![\Gamma \vdash M_2 : \tau]\!], \mathcal{E}[\![\Gamma \vdash M_3 : \tau]\!] \rangle \\
\mathcal{E}[\![\Gamma \vdash \texttt{fix}\ M : \tau]\!] &\triangleq \text{fix} \circ \text{unlift}(\mathcal{E}[\![\Gamma \vdash M : \tau \rightarrow \tau]\!]) \\
\mathcal{E}[\![\Gamma \vdash (M_1, M_2) : \tau_1 \times \tau_2]\!] &\triangleq \text{lift}_C \circ \langle \mathcal{E}[\![\Gamma \vdash M_1 : \tau_1]\!], \mathcal{E}[\![\Gamma \vdash M_2 : \tau_2]\!]) \rangle \\
\mathcal{E}[\![\texttt{fst}\ \Gamma \vdash M : \tau_1]\!] &\triangleq \pi_1 \circ \text{unlift} \circ \mathcal{E}[\![\Gamma \vdash M : \tau_1 \times \tau_2]\!] \\
\mathcal{E}[\![\texttt{snd}\ \Gamma \vdash M : \tau_2]\!] &\triangleq \pi_2 \circ \text{unlift} \circ \mathcal{E}[\![\Gamma \vdash M : \tau_1 \times \tau_2]\!] \\
\mathcal{E}[\![\Gamma \vdash r : \texttt{Real}]\!] &\triangleq \text{lift}_C \circ \text{const} \circ \Upsilon(r) \\
\mathcal{E}[\![\Gamma \vdash rop : \texttt{Real}^n \rightarrow \texttt{Real}]\!] &\triangleq \text{lift}_C \circ \text{const} \circ \Upsilon(rop) \\
\mathcal{E}[\![\Gamma \vdash dist : \texttt{Dist}\ \tau]\!] &\triangleq \text{lift}_C \circ \text{const} \circ \Upsilon(dist) \\
\mathcal{E}[\![\Gamma \vdash \texttt{return}\ M : \texttt{Dist}\ \tau]\!] &\triangleq \langle \text{det} \circ f, \eta \circ f \rangle \text{ where } f = \mathcal{E}[\![\Gamma \vdash M : \tau]\!] \\
\mathcal{E}[\![\Gamma \vdash x \leftarrow M_1\ ;\ M_2 : \texttt{Dist}\ \tau_2]\!] &\triangleq \langle \text{samp} \circ \langle \pi_1 \circ f, \pi_1 \circ \text{curry}(g) \rangle, \\
&\qquad (\pi_2 \circ f) \succ_{\flat} (\pi_2 \circ g) \rangle \\
&\qquad \text{where } f = \mathcal{E}[\![\Gamma \vdash M_1 : \texttt{Dist}\ \tau_1]\!] \\
&\qquad \text{where } g = \mathcal{E}[\![\Gamma, x : \tau_1 \vdash M_2 : \tau_2]\!]
\end{aligned}
$$

Figure 3.6 The denotational semantics of λ_{CD} is given by induction on the typing derivation (semantics of additional constructs are shaded). The structure of the semantics is similar to one where we use CPOs. Υ is a global environment used to interpret constants. The function π_x projects the variable x from the environment.

Well-formed global environment To ensure that we do not introduce non-computable constants into λ_{CD} (*e.g.*, non-computable operations on reals *rop*) and that the constants have the appropriate types, the global environment Υ should be well formed with respect to the global context Ψ. To distinguish the semantic value obtained from a global environment lookup from the syntax, we will put a bar over the constant (*e.g.*, $\Upsilon(r) = \bar{r}$) to refer to the semantic value. We say that Υ is well formed with respect to Ψ, written $\Psi \vdash \Upsilon$, if the conditions below hold.

(*real-wf*) For any $r \in \text{dom}(\Psi)$, $\Upsilon(r)$ is the realizer of a real $\bar{r}$ when $\Psi(r) = \texttt{Real}$.
(*dist-wf*) For any $dist \in \text{dom}(\Psi)$, $\Upsilon(dist)$ is the name of a pair $\overline{dist}$ that realizes a

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

sampler over values in $\mathcal{V}[\![\tau]\!]$ and the corresponding distribution when $\Psi(dist) =$ `Dist` τ.

(*rop-wf*) For any $rop \in \mathrm{dom}(\Psi)$, the corresponding semantic function $\overline{rop}$ is strict, continuous on its domain, and a n-ary real-valued function on reals when $\Psi(rop) = \mathtt{Real}^n \to \mathtt{Real}$.

Denotation function and sampling The denotation of terms corresponding to the PCF fragment are standard. Hence, we will focus on the constructs λ_{CD} introduces. The denotation of a constant real r is a global environment lookup.

$$\mathcal{E}[\![\Gamma \vdash r : \mathtt{Real}]\!] \triangleq \mathrm{lift}_C \circ \mathrm{const} \circ \Upsilon(r)$$

By the well-formedness of the global environment, $\Upsilon(r)$ will have a realizer. Likewise, the denotation of a primitive function on reals *rop* is a global environment lookup and corresponds to a representation of the code implementing the function.

$$\mathcal{E}[\![\Gamma \vdash rop : \mathtt{Real}^n \to \mathtt{Real}]\!] \triangleq \mathrm{lift}_C \circ \mathrm{const} \circ \Upsilon(rop)$$

The well-formedness of the global environment Υ enforces these conditions. Our next task is to explain the denotation of distribution constructs in λ_{CD}.

As a reminder, the interpretation of types is a pair of a sampler and the distribution that it realizes. As we will see shortly, the semantics of the sampling component and the semantics of the distribution component do not depend on one another (besides the fact that we want the distribution to be realized by the sampler). Hence, we could have given two different semantics and related them. Nevertheless, in this form, we will obtain that the valuation is the pushforward along the sampler, and consequently, make the connection between what is given by a distributional semantics and what was implemented in the sampling library. We walk through the distribution constructs now.

The denotation of a constant primitive distribution *dist* is a global environment lookup. Note that the interpretation of `Dist` τ is a pair of a sampler and valuation so the lookup should also produce a pair.

$$\mathcal{E}[\![\Gamma \vdash dist : \mathtt{Dist}\ \tau]\!] \triangleq \mathrm{lift}_C \circ \mathrm{const} \circ \Upsilon(dist)$$

The denotation of `return` M produces a pair of a sampler that ignores the input bit-randomness and a point mass valuation centered at M.

$$\mathcal{E}[\![\Gamma \vdash \mathtt{return}\ M : \mathtt{Dist}\ \tau]\!] \triangleq \langle \det \circ f, \eta \circ f \rangle \text{ where } f = \mathcal{E}[\![\Gamma \vdash M : \tau]\!]$$

The meaning of $x \leftarrow M\ ;\ N$ also gives a sampler and a valuation.

$$\mathcal{E}[\![\Gamma \vdash x \leftarrow M_1\ ;\ M_2 : \mathtt{Dist}\ \tau_2]\!] \triangleq \langle \mathrm{samp} \circ \langle \pi_1 \circ f, \pi_1 \circ \mathrm{curry}(g) \rangle, (\pi_2 \circ f) \succ_\flat (\pi_2 \circ g) \rangle$$

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

where $f = \mathcal{E}[\![\Gamma \vdash M_1 : \texttt{Dist}\ \tau_1]\!]$ and $g = \mathcal{E}[\![\Gamma, x : \tau_1 \vdash M_2 : \tau_2]\!]$. Under the sampling view, we use samp to compose the sampler obtained by $\pi_1 \circ f$ with the function $\pi_1 \circ g$. Under the valuation component, we reweigh $\pi_2 \circ g$ according to the valuation $\pi_2 \circ f$ using monad bind $\succ_\flat$ from $\boldsymbol{P}$. We can check that the valuation given by the semantics is indeed the pushforward along the sampler.

Proposition 3.37 (Push) *Let D and E be countably based topological predomains (qcb$_0$ spaces more generally).*
(i) $\mathrm{psh}_D(\mathrm{det}(d)) = \eta(d)$ *for any* $d \in D$.
(ii) $\mathrm{psh}_E(\mathrm{samp}(s)(f)) = \mathrm{psh}_D(s) \succ_\flat v \mapsto \mathrm{psh}_E(f(v))$ *for any* $s \in \boldsymbol{S}(D)$ *and* $f : D \Rightarrow \boldsymbol{S}(E)$.

In the case of bind (the second item), it is necessary that the split operation used in samp (Section 3.4.2) produces an independent stream of bits.

We end by checking that the expression denotation function is well defined.

Proposition 3.38 (Well defined) *The expression denotation function* $\mathcal{E}[\![\cdot]\!]$ *is well defined, i.e.,* $\mathcal{E}[\![\Gamma \vdash M : \tau]\!] : \mathcal{V}[\![\Gamma]\!] \Rightarrow \mathcal{V}[\![\tau]\!]$ *for any well-typed term* $\Gamma \vdash M : \tau$ *and well-formed global environment* $\Psi \vdash \Upsilon$.

The structure of the argument showing that the expression denotation function is well defined is similar to the argument for showing that the CPO semantics of PCF is well defined. The interesting cases correspond to `return` M and $x \leftarrow M_1 \ ; \ M_2$ where we need to relate the sampling component with the valuation it denotes, which is given by Proposition 3.37.

3.5.3 Reasoning About Programs

We now return to resolving some semantic issues that were raised when we used the library to implement distributions. Throughout this section, we overload $\mathcal{E}[\![\Gamma \vdash M : \texttt{Dist}\ \tau]\!]$ to mean $\pi_2 \circ \mathcal{E}[\![\Gamma \vdash M : \texttt{Dist}\ \tau]\!]$ so that it just provides the distributional view. As shorthand, we write $\mathcal{E}[\![\cdot]\!]\rho$ instead of $\mathcal{E}[\![\cdot]\!](\rho)$ where the meta-variable ρ ranges over environments.

Reasoning about distributions We first show that the encoding of the standard geometric distribution is correct. Let μ_B be an unbiased Bernoulli distribution and

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

μ^n correspond to n un-foldings of `stdGeometric`:

$$\mathcal{E}[\![\texttt{stdGeometric}]\!]\rho(U) = \sup_{n\in\mathbb{N}} \int \left(v \mapsto \begin{cases} \mathbb{1}_U(1) & v = t \\ \int w \mapsto \mathbb{1}_U(w+1)d\mu^n & v = f \end{cases} \right) d\mu_B$$
$$= \sup_{n\in\mathbb{N}}(\mathbb{1}_U(1)\frac{1}{2} + \sum_{w=0}^{\infty} \mathbb{1}_U(w+1)\mu^n(\{w\}))$$
$$= \mathbb{1}_U(1)\frac{1}{2} + \sum_{w=0}^{\infty} \mathbb{1}_U(w+1)(\sup_{n\in\mathbb{N}} \mu^n(\{w\}))\,.$$

By induction on n, we can show that μ^n is the measure

$$\mu^n = \{0\} \mapsto 0, \{1\} \mapsto (1/2), \ldots, \{n\} \mapsto (1/2)^n \ .$$

Hence, we can conclude that $\sup_{n\in\mathbb{N}} \mu^n$ is a geometric distribution and that the encoding of `stdGeometric` is correct (for any environment ρ).

Primitive functions In our encoding of the standard normal distribution via the Marsaglia polar transformation, we used $<$ as if it had a return type of `Bool` even though equality on reals is not computable. Indeed, the well-formedness conditions imposed on the global environment would disallow $<$ at the current type. To resolve the semantics of $<$, we can think in terms of an implementation. In particular, we can encode $<$ as dovetailing computations that semi-decides $x < y$ (*i.e.*, return `()` if $x < y$ and diverge otherwise) and semi-decides $x > y$ (*i.e.*, returns `()` if $x > y$ and diverge otherwise)). On the case of equality, which occurs with probability 0 in the Marsaglia polar transform, the function diverges.

Partiality and divergence We investigate the semantics of divergence more closely now. For convenience, we repeat the two expressions from Section 3.3 that provided two differing notions of divergence below.

```
botSamp :: (CMetrizable a) => Samp (Approx a)
botSamp = botSamp

botSampBot :: (CMetrizable a) => Samp (Approx a)
botSampBot = mkSamp (\_ -> bot)
    where bot = bot
```

In the former, we obtain the bottom valuation, which assigns 0 mass to every open set. This corresponds to the sampling function $u \in \mathbf{2}^\omega \mapsto \bot$ and can be interpreted as failing to provide a sampler. In the latter, we obtain the valuation that assigns 0 mass to every open set, except for the set $\{\lfloor X \rfloor \cup \bot\}$ which is assigned mass 1. This corresponds to the sampling function $u \in \mathbf{2}^\omega \mapsto \lfloor \bot \rfloor$ and can be interpreted as providing a sampling function that fails to produce a sample.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

As before, we can check that laziness works in the appropriate manner by selectively ignoring the results of draws from the distributions above.

```
alwaysDiv :: Samp Real
alwaysDiv = do
  _ <- botSamp ::
         Samp Real
  stdUniform
```

```
neverDiv :: Samp Real
neverDiv = do
  _ <- botSampBot ::
         Samp Real
  stdUniform
```

We can check that the denotation of the former is equivalent to that of `botSamp`:

$$\begin{aligned}\mathcal{E}[\![\texttt{alwaysDiv}]\!]\rho(U) &= \int (\cdot \mapsto \mu_{\mathcal{U}}(U)) \; d\mathcal{E}[\![\texttt{botSamp}]\!]\rho \\ &= 0\end{aligned}$$

where $\mu_{\mathcal{U}}$ is the standard uniform distribution. Note that $\mathcal{E}[\![\texttt{botSamp}]\!]\rho$ maps every open set to 0 so the integral is 0 as well. However, the denotation of the latter is equivalent to that of `stdUniform`:

$$\begin{aligned}\mathcal{E}[\![\texttt{neverDiv}]\!]\rho(U) &= \int (\cdot \mapsto \mu_{\mathcal{U}}(U)) \; d\mathcal{E}[\![\texttt{botSampBot}]\!]\rho \\ &= \sup_{s \text{ simple}} \left\{ \int s \; d\mathcal{E}[\![\texttt{botSampBot}]\!]\rho \mid s \le \cdot \mapsto \mu_{\mathcal{U}}(U) \right\} \\ &= \mu_{\mathcal{U}}(U)\,.\end{aligned}$$

As a reminder, $\mathcal{E}[\![\texttt{botSampBot}]\!]\rho(U) = 1$ when $U = \mathcal{V}[\![\texttt{Real}]\!]$. Hence, the integral takes its largest value on the simple function[33] $\mu_{\mathcal{U}}(U)\,\mathbb{1}_{\mathcal{V}[\![\texttt{Real}]\!]}(\cdot)$.

As a final example, consider the program below that uses a coin flip to determine its diverging behavior.

```
maybeBot :: Samp Bool
maybeBot = do
  b <- stdBernoulli
  if b then return bot else stdBernoulli
```

Intuitively, this distribution returns a *sampler that always generates diverging samples* with probability $1/2$ and returns an unbiased Bernoulli distribution with probability $1/2$. If we changed `return bot` to `botSamp` as below

```
maybeBot' :: Samp Bool
maybeBot' = do
  b <- stdBernoulli
  if b then bot else stdBernoulli
```

then the semantics would change to a distribution that returns a *diverging sampler* with probability $1/2$ and an unbiased Bernoulli distribution with probability $1/2$.

[33] As a reminder, a simple function in our context is a linear combination of indicator functions on open sets.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

Independence and commutativity In Section 3.3.2, we saw that we could not argue that two distributions that commuted the order in which we sampled independent normal distributions were equivalent. As a reminder, the issue was that commuting the order of sampling meant that the underlying random bit-stream was consumed in a different order. Consequently, the values produced by the two terms may be different. However, as the semantics we just saw relates the sampling view with the distributional view by construction, we can easily see that these two terms will be distributionally equivalent by Fubini.

3.6 Bayesian Inference

What are the implications of taking a computable viewpoint for Bayesian inference? In this section, we discuss the implications of taking a computable viewpoint for Bayesian inference. Perhaps surprisingly, one can show that conditioning is not computable in general. Nevertheless, conditioning in practical settings does not run into these pathologies. It will be important for probabilistic programming languages to support conditioning in these cases. Note that these results say nothing about the efficiency of inference. In practice, we will still need approximate inference algorithms to compute conditional distributions.

3.6.1 Conditioning is not Computable

Figure 3.7 gives an encoding in λ_{CD} of an example by Ackerman et al. (2011) that shows that conditioning is not always computable. Similar to other results

```
nonComp :: Samp (Nat, Real)
nonComp = do
  n <- geometric (1/2)
  c <- bernoulli (1/3)
  u <- uniform 0 1
  v <- uniform 0 1
  x <- return (mkApprox
               (\k -> select u v c k (tmHaltsIn n)))
  return (n, x)
    where select u v c k m
            | m > k = nthApprox v k
            | m == k = if c then 1 else 0
            | m < k = nthApprox u (k - m - 1)
```

Figure 3.7 A Haskell encoding of a counter-example given by Ackerman et al. (2011) that shows that conditioning is not always computable. The function `tmHaltsIn` in the code outputs the number of steps the n-th Turing machine halts in or ∞ (for diverges) if the n-th Turing machine does not halt. The idea is that an algorithm that could compute this conditional distribution would imply a decision procedure for the Halting problem.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

in computability theory, the example demonstrates that an algorithm computing the conditional distribution would also solve the Halting problem. The function `tmHaltsIn` accepts a natural n specifying the n-th Turing machine and outputs the number of steps the n-th Turing machine halts in or ∞ (for diverges) if the n-the Turing machine does not halt. Upon inspection, we see the function `nthApprox` produces the binary expansion (as a dyadic rational) of a real, using `tmHaltsIn` to select different bits of the binary expansion of u or v, or the bit c depending on whether the n-th Turing machine halts within k steps or not.

Consider computing the conditional distribution $\mathbb{P}(\mathbf{N} \mid \mathbf{X})$, where the random variable $\mathbf{N}$ corresponds to the program variable n and $\mathbf{X}$ to x. Thus, computing the conditional distribution $\mathbb{P}(\mathbf{N} \mid \mathbf{X})$ corresponds to determining the value of the program variable n given the value of the program variable x. We informally discuss why this distribution is not computable now. Observe that (1) the value of x depends on the value of n—whether the n-th Turing machine halts within k steps or not for every k (hence whether the n-th Turing machine halts or not)—and (2) the geometric distribution is supported on $\mathbb{N}\backslash\{0\}$ so we need to consider every Turing machine. Consequently, we would require a decision procedure for the Halting problem in order to compute the posterior distribution on n. Thus, `nonComp` encodes a computable distribution $\mathbb{P}(\mathbf{N}, \mathbf{X})$ whose conditional $\mathbb{P}(\mathbf{N} \mid \mathbf{X})$ is not computable. We refer the reader to the full proof (see Ackerman et al., 2011) for more details.

3.6.2 Conditioning is Computable

Now, we add conditioning as a library to λ_{CD} (Figure 3.8). λ_{CD} provides only a restricted conditioning operation `obsDens`, which requires a conditional density. We will see that the computability of `obsDens` corresponds to an effective version of Bayes' rule. We have given only one conditioning primitive here, but it is possible to identify other situations where conditioning is computable and add those to the conditioning library. For example, conditioning on positive probability events is computable (see Galatolo et al., 2010, Prop. 3.1.2).

The library provides the conditioning operation `obsDens`, which enables us to condition on *continuous*-valued data when a bounded and computable conditional density is available.

Proposition 3.39 (Ackerman et al., 2011, Cor. 8.8). *Let $\boldsymbol{U}$ be a U-valued random variable, $\boldsymbol{V}$ be a V-valued random variable, and $\boldsymbol{Y}$ be Y-valued random variable, where $\boldsymbol{Y}$ is independent of $\boldsymbol{V}$ given $\boldsymbol{U}$. Let $\boldsymbol{U}$, $\boldsymbol{V}$, and $\boldsymbol{Y}$ be computable. Moreover, let $p_{\boldsymbol{Y}|\boldsymbol{U}}(y \mid u)$ be a conditional density of $\boldsymbol{Y}$ given $\boldsymbol{U}$ that is bounded and computable. Then the conditional distribution $\mathbb{P}[(\boldsymbol{U}, \boldsymbol{V}) \mid \boldsymbol{Y}]$ is computable.*

The bounded and computable conditional density enables the following integral

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

```
module CondLib (BndDens, obsDens) where
import ApproxLib
import CompDistLib
import RealLib

newtype BndDens a b =
    BndDens { getBndDens :: (Approx a -> Approx b -> Real, Rat)
           }

-- Requires bounded and computable density
obsDens :: forall u v y.
    (CMetrizable u, CMetrizable v, CMetrizable y) =>
    Samp (Approx (u, v)) -> BndDens u y -> Approx y -> Samp (
        Approx (u, v))

-- Extend with more conditioning operators below ...
```

Figure 3.8 An interface for conditioning (module `CondLib`). The function `obsDens` enables conditioning on continuous-valued data when a bounded and computable conditional density is available.

to be computed, which is in essence Bayes' rule. A version of the conditional distribution $\mathbb{P}((\boldsymbol{U},\boldsymbol{V}) \mid \boldsymbol{Y})$ is

$$\kappa_{(\boldsymbol{U},\boldsymbol{V})|\boldsymbol{Y}}(y,B) = \frac{\int_B p_{\boldsymbol{Y}|\boldsymbol{U}}(y \mid u)\, d\mu_{(\boldsymbol{U},\boldsymbol{V})}}{\int p_{\boldsymbol{Y}|\boldsymbol{U}}(y \mid u)\, d\mu_{(\boldsymbol{U},\boldsymbol{V})}}$$

where B is a Borel set in the space associated with $U \times V$ and $\mu_{(\boldsymbol{U},\boldsymbol{V})}$ is the joint distribution of $\boldsymbol{U}$ and $\boldsymbol{V}$.[34]

Another interpretation of the restricted situation is that our observations have been corrupted by independent smooth noise (Ackerman et al., 2011, Cor. 8.9). To see this, consider the following generative model:

$$\begin{aligned}(\boldsymbol{U},\boldsymbol{V}) &\sim \mu_{(\boldsymbol{U},\boldsymbol{V})}\\ \boldsymbol{N} &\sim \mu_{\text{noise}}\\ \boldsymbol{Y} &= \boldsymbol{U} + \boldsymbol{N}\end{aligned}$$

where μ_{noise} has density $p_{\boldsymbol{N}}(\cdot)$. The random variable $\boldsymbol{U}$ can be interpreted as the ideal model of how the data was generated and the random variable $\boldsymbol{V}$ can be interpreted as the model parameters. The random variable $\boldsymbol{Y}$ can then be interpreted as the data we observe that is smoothed by the noise $\boldsymbol{N}$ so that $p_{\boldsymbol{Y}|\boldsymbol{U}}(y \mid u) = p_{\boldsymbol{N}}(y - u)$. Notice that the model $(\boldsymbol{U},\boldsymbol{V})$ is not required to have a density and can be an arbitrary

[34] As a reminder, $p_{\boldsymbol{Y}|(\boldsymbol{U},\boldsymbol{V})}(y \mid u, v) = p_{\boldsymbol{Y}|\boldsymbol{U}}(y \mid u)$ due to the conditional independence of $\boldsymbol{Y}$ and $\boldsymbol{V}$ given $\boldsymbol{U}$. Hence, the conditional density $p_{\boldsymbol{Y}|\boldsymbol{U}}(y \mid u)$ in the integral written more precisely is $(u, v) \mapsto p_{\boldsymbol{Y}|\boldsymbol{U}}(y \mid u)$.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

computable distribution. The idea is that we condition on $\boldsymbol{Y}$ (*i.e.*, the smoothed data) as opposed to $\boldsymbol{U}$ (*i.e.*, the ideal data) when we compute the posterior distribution for the model parameters $\boldsymbol{V}$.[35] Indeed, probabilistic programming systems proposed by the machine learning community impose a similar restriction (*e.g.*, see Goodman et al., 2008; Wood et al., 2014).

Now, we describe `obsDens`, starting with its type signature. Let the type `BndDens` τ σ represent a bounded computable density:

```
newtype BndDens a b =
    BndDens { getBndDens :: (Approx a -> Approx b -> Real, Rat)
            }
```

Conditioning thus takes a samplable distribution, a bounded computable density describing how observations have been corrupted, and returns a samplable distribution representing the conditional. In the context of Bayesian inference, it does not make sense to condition distributions such as `maybeBot` that diverge with positive probability. Hence, we do not give semantics to conditioning on those distributions.

The implementation of `obsDens` is in essence a λ_{CD} program that implements the proof that conditioning is computable in this restricted setting. This is possible because results in computability theory have computable realizers.[36]

```
obsDens :: forall u v y.
    (CMetrizable u, CMetrizable v, CMetrizable y) =>
    Samp (Approx (u, v)) -> BndDens u y -> Approx y -> Samp (
       Approx (u, v))
obsDens dist (BndDens (dens, bnd)) d =
    let f :: Approx (u, v) -> Real = \x -> dens (approxFst x) d
        mu :: Prob (u, v) = sampToComp dist
        nu :: Prob (u, v) = \bs ->
              let num   = integrateBndDom mu f bnd bs
                  denom = integrateBnd mu f bnd
              in map fst (cauchyToLU (num / denom))
    in
      compToSamp nu
```

The parameter `dist` corresponds to the joint distribution of the model (both model parameters and likelihood), `dens` corresponds to a bounded conditional density

[35] As an illustrative example, consider the following situation where we hope to model how a scene in an image is constructed so we can identify objects in the image (see Kulkarni et al., 2015, for a probabilistic programming language designed for scene perception). The model parameters $\boldsymbol{V}$ contain all the information describing the objects in the scene, including their optical properties and their positions. The resulting image $\boldsymbol{U}$ is then rendered by a graphics engine. The caveat is that the graphics engine uses an enumeration of the Halting set to add artifacts to the image so it is pathological. (Thus, this distribution is a version of `nonComp`.) Instead of attempting to compute the posterior distribution $\mathbb{P}(\boldsymbol{U} \mid \boldsymbol{V})$, which is not computable, we smooth out the rendered image $\boldsymbol{U}$ with some noise given by $p_{\boldsymbol{Y}|\boldsymbol{U}}(y \mid u)$. In other words, we apply some filtering to the image $\boldsymbol{U}$ so we obtain an image $\boldsymbol{Y}$ free of artifacts introduced by the pathological graphics engine. The posterior $\mathbb{P}((\boldsymbol{U}, \boldsymbol{V}) \mid \boldsymbol{V})$ is then computable. In this example, the posterior would give the positions and optical properties of objects given an image so it could be used in computer vision applications.

[36] That is, we implement the Type-2 machine code as a Haskell program.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

describing how observation of data has been corrupted by independent noise, and `d` is the observed data. Next, we informally describe the undefined functions in the sketch. The function `approxFst` projects out the first component of a product of approximations. The functions `sampToComp` and `compToSamp` witness the computable isomorphism between samplable and computable distributions.[37] The functions `integrateBndDom` and `integrateBnd` compute an integral (see Hoyrup and Rojas, 2009, Prop. 4.3.1), and correspond to an effective Lebesgue integral. `cauchyToLU` converts a Cauchy description of a computable real into an enumeration of lower and upper bounds.

Because `obsDens` works with conditional densities, we do not need to worry about the Borel paradox. The Borel paradox shows that we can obtain different conditional distributions when conditioning on probability zero events (*e.g.*, see Rao and Swift, 2006). To illustrate this, suppose that $\mathbf{X}$ and $\mathbf{Y}$ are two independent random variables with standard normal distributions. We can ask a (classic) question: "What is the conditional distribution of $\mathbf{Y}$ given that $\mathbf{X} = \mathbf{Y}$?"

In statistics, the appropriate response is to notice that the question as posed is ill-formed—one cannot condition on a measure zero event. The well-posed formulation is to define an auxiliary random variable $\mathbf{Z}$ and condition on a constant. For instance, $\mathbf{Z} = \mathbf{X} - \mathbf{Y}$ conditioned on $\mathbf{Z} = 0$, $\mathbf{Z} = \mathbf{Y}/\mathbf{X}$ conditioned on $\mathbf{Z} = 1$, and $\mathbf{Z} = \mathbb{I}_{Y=X}$ conditioned on $\mathbf{Z} = 1$. Remarkably, all three versions lead to different answers (Proschan and Presnell, 1998).

A probabilistic programming language that does not provide a notion of random variable such as λ_{CD} will need an alternative method of addressing this issue. Type-2 computability provides a straight-forward answer—it is not possible to create a boolean value that distinguishes two probability zero events in λ_{CD}. For instance, the operator == implementing equality on reals returns false if two reals are provably not-equal and diverges otherwise because equality is not decidable.

3.7 Summary and Further Directions

We hope to have shown that we do not need to sacrifice traditional notions of computation when modeling reals and continuous distributions by keeping their *representations* in mind. The simple observation is that we can "program" them in a general-purpose programming language. With this in mind, we can now ask a basic question: "What does it mean for a probabilistic programming language to be Turing-complete?" From the perspective of Type-2 computability, one answer is that such a language can express all *Type-2 computable distributions*, analogous to

[37] The computable isomorphism relies on the distributions being full-measure. The algorithm is undefined otherwise.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

how a Turing-complete language can express all computable functions. Indeed, this resolution is somewhat tautological!

This answer raises another interesting question related to full-abstraction and universality[38] of probabilistic programs. In the standard setting of PCF, one approach to the full-abstraction problem is to add *parallel or* `por` to the language so that the operational behavior coincides with the denotational semantics. Additionally adding a searching operator `exists` means that all computable functions will be definable. One may wonder, if an analogous result holds for probabilistic programs. In particular, a universality result would crystallize the thought that Turing-complete probabilistic programming languages express Type-2 computable distributions.

As we are now back on familiar grounds with regards to computability, we can turn our attention to the *design* of probabilistic programming languages. The design of such languages will demand more from a semantics of probabilistic programs. For example, for the purposes of automating Bayesian inference, it is crucial that the inference procedure be *efficient* (and not simply computable). One direction is to find compilation strategies that can efficiently realize Type-2 computable distributions or approximate them (for some notion of approximation) using floating point numbers. Another direction is to consider alternative language designs (in addition to PCF with a probability monad) and the corresponding structures that we will need to model these languages.

Acknowledgments

We thank our anonymous reviewers for their helpful comments and feedback.

References

Abramsky, Samson, and Jung, Achim. 1994. Domain Theory. Pages 1–168 of: Abramsky, Samson, Gabbay, Dov M, and Maibaum, T S E (eds), *Handbook of Logic in Computer Science*, vol. 3. Oxford University Press.

Ackerman, Nathanael Leedom, Freer, Cameron E, and Roy, Daniel M. 2011. Noncomputable Conditional Distributions. Pages 107–116 of: *Proceedings of the 26th Annual Symposium on Logic in Computer Science*. IEEE.

Battenfeld, Ingo. 2004. *A Category of Topological Predomains*. M.Phil. thesis, TU Darmstadt.

Battenfeld, Ingo. 2008. *Topological Domain Theory*. Ph.D. thesis, University of Edinburgh.

Battenfeld, Ingo, Schröder, Matthias, and Simpson, Alex. 2006. Compactly generated domain theory. *Mathematical Structures in Computer Science*, **16**(2), 141–161.

[38] A programming language is universal if all computable elements in the domain of interpretation are definable.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

Battenfeld, Ingo, Schröder, Matthias, and Simpson, Alex. 2007. A Convenient Category of Domains. *Electronic Notes in Theoretical Computer Science*, **172**, 69–99.

Bauer, Andrej. 2000a. *The Realizability Approach to Computable Analysis and Topology*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

Bauer, Andrej. 2000b. *The Realizability Approach to Computable Analysis and Topology*. Ph.D. thesis, Carnegie Mellon University.

Bauer, Andrej. 2005. *Realizability as the connection between computable and constructive mathematics*. Unpublished lecture notes.

Bauer, Andrej, and Kavkler, Iztok. 2008. Implementing Real Numbers With RZ. *Electronic Notes in Theoretical Computer Science*, **202**, 365–384.

Birkedal, Lars. 1999. *Developing Theories of Types and Computability via Realizability*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University.

Borgström, Johannes, Gordon, Andrew D, Greenberg, Michael, Margetson, James, and Van Gael, Jurgen. 2011. Measure Transformer Semantics for Bayesian Machine Learning. Pages 77–96 of: *Programming Languages and Systems*. Springer.

Crubillé, Raphaëlle. 2018. Probabilistic Stable Functions on Discrete Cones are Power Series. Pages 275–284 of: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM.

Dal Lago, Ugo, and Zorzi, Margherita. 2012. Probabilistic operational semantics for the lambda calculus. *RAIRO-Theoretical Informatics and Applications*, **46**(3), 413–450.

Danos, Vincent, and Ehrhard, Thomas. 2011. Probabilistic coherence spaces as a model of higher-order probabilistic computation. *Information and Computation*, **209**(6), 966–991.

Durrett, Rick. 2010. *Probability: Theory and Examples*. 4 edn. Cambridge University Press.

Ehrhard, Thomas, Tasson, Christine, and Pagani, Michele. 2014. Probabilistic Coherence Spaces are Fully Abstract for Probabilistic PCF. Pages 309–320 of: *ACM SIGPLAN Notices*, vol. 49. ACM.

Ehrhard, Thomas, Pagani, Michele, and Tasson, Christine. 2018. Measurable Cones and Stable, Measurable Functions: A Model for Probabilistic Higher-Order Programming. *Proceedings of the ACM on Programming Languages*, **2**(POPL), 59.

Escardó, Martín Hötzel. 1996. PCF extended with real numbers. *Theoretical Computer Science*, **162**(1), 79–115.

Escardó, Martín Hötzel. 2004. Synthetic topology: of data types and classical spaces. *Electronic Notes in Theoretical Computer Science*, **87**, 21–156.

Escardó, Martín Hötzel, Lawson, Jimmie, and Simpson, Alex. 2004. Comparing Cartesian closed categories of (core) compactly generated spaces. *Topology and its Applications*, **143**(1), 105–145.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

Freer, Cameron E, and Roy, Daniel M. 2010. Posterior distributions are computable from predictive distributions. Pages 233–240 of: *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*. SAIS.

Galatolo, Stefano, Hoyrup, Mathieu, and Rojas, Cristóbal. 2010. Effective symbolic dynamics, random points, statistical behavior, complexity and entropy. *Information and Computation*, **208**(1), 23–41.

Giry, Michèle. 1982. A categorical approach to probability theory. Pages 68–85 of: *Categorical Aspects of Topology and Analysis*. Springer.

Goodman, Noah, Mansinghka, Vikash, Roy, Daniel M, Bonawitz, Keith, and Tenenbaum, Joshua B. 2008. Church: A language for generative models. Pages 220–229 of: *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*. AUAI.

Gunter, Carl A. 1992. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. The MIT Press.

Heunen, Chris, Kammar, Ohad, Staton, Sam, and Yang, Hongseok. 2017. A Convenient Category for Higher-Order Probability Theory. Pages 1–12 of: *Logic in Computer Science (LICS), 2017 32nd Annual ACM/IEEE Symposium on*. IEEE.

Hoyrup, Mathieu, and Rojas, Cristóbal. 2009. Computability of probability measures and Martin-Löf randomness over metric spaces. *Information and Computation*, **207**(7), 830–847.

Huang, Daniel, and Morrisett, Greg. 2016. An Application of Computable Distributions to the Semantics of Probabilistic Programming Languages. Pages 337–363 of: *Programming Languages and Systems*.

Huang, Daniel Eachern. 2017. *On Programming Languages for Probabilistic Modeling*. Ph.D. thesis, Harvard University.

Jones, Claire. 1989. *Probabilistic Non-determinism*. Ph.D. thesis, University of Edinburgh.

Kozen, Dexter. 1981. Semantics of Probabilistic Programs. *Journal of Computer and System Sciences*, **22**(3), 328–350.

Kulkarni, Tejas D, Kohli, Pushmeet, Tenenbaum, Joshua B, and Mansinghka, Vikash. 2015. Picture: A Probabilistic Programming Language for Scene Perception. Pages 4390–4399 of: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE.

Lambov, Branimir. 2007. RealLib: An Efficient Implementation of Exact Real Arithmetic. *Mathematical Structures in Computer Science*, **17**(1), 81–98.

Lietz, Peter. 2004. *From Constructive Mathematics to Computable Analysis via the Realizability Interpretation*. Ph.D. thesis, TU Darmstadt.

Longley, John R. 1995. *Realizability Toposes and Language Semantics*. Ph.D. thesis, University of Edinburgh.

Munkres, James R. 2000. *Topology*. 2 edn. Prentice Hall.

Panangaden, Prakash. 1999. The Category of Markov Kernels. *Electronic Notes in Theoretical Computer Science*, **22**, 171–187.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press

Park, Sungwoo, Pfenning, Frank, and Thrun, Sebastian. 2005. A Probabilistic Language Based Upon Sampling Functions. Pages 171–182 of: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

Peyton Jones, Simon, Reid, Alastair, Henderson, Fergus, Hoare, Tony, and Marlow, Simon. 1999. A Semantics for Imprecise Exceptions. Pages 25–36 of: *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM.

Proschan, Michael A, and Presnell, Brett. 1998. Expect the Unexpected from Conditional Expectation. *The American Statistician*, **52**(3), 248–252.

Rao, Malempati M, and Swift, Randall J. 2006. *Probability theory with applications*. 2 edn. Mathematics and Its Applications, vol. 582. Springer.

Saheb-Djahromi, Nasser. 1978. Probabilistic LCF. *Mathematical Foundations of Computer Science*, **64**, 442–451.

Schröder, Matthias. 2007. Admissible Representations of Probability Measures. *Electronic Notes in Theoretical Computer Science*, **167**, 61–78.

Sipser, Michael. 2012. *Introduction to the Theory of Computation*. 3 edn. Cengage Learning.

Staton, Sam. 2017. Commutative Semantics for Probabilistic Programming. Pages 855–879 of: *European Symposium on Programming*. Springer.

Streicher, Thomas. 2008. *Realizability*. http://www.mathematik.tu-darmstadt.de/~streicher/REAL/REAL.pdf. Course Lecture Notes.

Vákár, Matthijs, Kammar, Ohad, and Staton, Sam. 2019. A Domain Theory for Statistical Probabilistic Programming. *Proceedings of the ACM on Programming Languages*, **3**(POPL), 36.

Weihrauch, Klaus. 2000. *Computable Analysis: An Introduction*. 2000 edn. Texts in Theoretical Computer Science. An EATCS Series. Springer.

Wood, Frank, van de Meent, Jan Willem, and Mansinghka, Vikash. 2014. A new approach to probabilistic programming inference. Pages 2–46 of: *Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*. SAIS.

https://doi.org/10.1017/9781108770750.004 Published online by Cambridge University Press