An application of computable distributions to the semantics of probabilistic programming languages

Daniel Huang[1] and Greg Morrisett[2]

[1] Harvard SEAS, Cambridge MA
dehuang@fas.harvard.edu
[2] Cornell University, Ithaca NY
greg.morrisett@cornell.edu

Abstract. Most probabilistic programming languages for Bayesian inference give either operational semantics in terms of sampling, or denotational semantics in terms of measure-theoretic distributions. It is important that we can relate the two, given that practitioners often reason both analytically (*e.g.*, density) as well as algorithmically (*i.e.*, in terms of sampling) about distributions. In this paper, we give denotational semantics to a functional language extended with continuous distributions and show that by restricting attention to *computable distributions*, we can realize a corresponding sampling semantics.

Keywords: Probabilistic programs, computable distributions, semantics

1 Introduction

Probabilistic programming for Bayesian inference hopes to simplify probabilistic modeling by (1) providing a programming language for users to formally specify probabilistic models and (2) automating the task of inferring model parameters given observed data.

A natural interpretation of these probabilistic programs is as a sampler (*e.g.*, [9, 35]). In essence, we can view a probabilistic program as describing a sampling procedure encoding how one believes observed data is generated. The flexibility of sampling has inspired embeddings of probabilistic primitives in full-fledged programming languages, including Scheme and Scala (*e.g.*, [9, 23]). However, languages based on sampling often avoid the basic question of what it means to sample from a continuous distribution. For instance, these languages either have semantics given by an implementation in a host-language (*e.g.*, [9]), or use an abstract machine that assumes reals and primitive continuous distributions [21].

Another approach grounds the semantics of probabilistic programs in measure theory, which is the foundation of probability theory (*e.g.*, [3, 29]). Measure theory provides a rigorous definition of conditioning, and a uniform treatment

of discrete and continuous distributions. For instance, measure-theoretic probability makes sense of why the probability of obtaining any sample from an (absolutely) continuous distribution is zero, as well what it means to observe a probability zero event. These situations are encountered frequently in practice, as many models incorporate both continuous distributions and observation of real-valued data. Consequently, measure-theoretic semantics have been proposed as a generalization of sampling semantics (*e.g.*, [3]). However, this approach also has its drawbacks. First, measure theory has been developed without programming language constructs in mind (*e.g.*, recursion and higher-order functions), unlike standard denotational semantics. Hence, languages based on measure theory often omit language features (*e.g.*, [3]) or develop new meta-theory (*e.g.*, [29]). Second, measure-theoretic semantics must also develop a corresponding operational theory. For instance, Toronto *et al.* [29] give denotational semantics and later show how to implement a sound approximation of their ideal semantics because it is not directly implementable.

In this paper, we address the shortcomings of these existing approaches, and give denotational semantics to a functional language with continuous distributions that uses largely standard semantic constructs, and make the connection to a faithful sampling implementation. Our approach is motivated by *Type-2 computable distributions*, which admit *Type-2 computable* sampling procedures.[3] Computable distributions have been defined and studied in the context of Type-2 (Turing) machines and algorithmic randomness (*e.g.*, [6, 31, 7]). Their implications for probabilistic programs have also been hinted at in the literature (*e.g.*, [1, 5]). Hence, we will recast these ideas in the context of high-level probabilistic languages for Bayesian inference.

There are advantages to giving semantics with computable distributions in mind, instead of directly using measure-theoretic distributions. First, every computable distribution admits a *sampling algorithm*, which operates on input bit-randomness (*e.g.*, a stream of coin flips) instead of requiring black-box primitives that generate Uniformly distributed values. Indeed, a computable distribution is completely characterized by a *sampling algorithm* (see Prop. 1), which reflects the intuition that we can express a distribution by describing how to sample from it. Second, we can use results from computability theory to guide the design of such a language. In particular, Ackerman *et al.* show that computing a conditional distribution is akin to solving the Halting problem [1]. Thus, our language provides only conditioning operators for restricted settings. Finally, computable distributions can be represented as ordinary program values. This means that we can use standard programming language meta-theory, and it enables us to scale our approach to probabilistic languages embedded in full-fledged programming languages proposed by the machine learning community (*e.g.*, [9, 23]).

However, our approach has some limitations. As we already mentioned above, conditioning is not computable in general so we do not give semantics to a

[3] Because we will use the phrase "Type-2 computable" frequently, we will sometimes abbreviate it to just "computable" when it is clear from context that we are referring to Type-2 computability.

generic conditioning operator (see Sec. 5). Nonetheless, the situation where we do give semantics to conditioning corresponds to an effective version of Bayes rule, which is central to Bayesian inference in practice. Second, even though our approach gives realizable semantics, it is not necessarily efficient because algorithms that operate on computable distributions compute by enumeration. It would be interesting to see if we can efficiently implement algorithms that compute approximately with computable distributions, but this is not in scope for this paper.

2 The basic idea

We begin by expanding upon the issues involved in giving semantics to probabilistic programming languages. In order to focus on the probabilistic aspect, we informally consider simple, first-order languages without recursion, extended with discrete and continuous distributions.

We start with adding discrete distributions with finite support to a programming language (*e.g.*, [25]), and illustrate how sampling and distribution semantics can coincide. A distribution with finite support assigns positive probability to a finite number of values in its domain. We can interpret a type $\texttt{Dist}\ \alpha$ as a *probability mass function*, which maps values of type α (written $\mathcal{V}[\![\alpha]\!]$) to a probability in the interval $[0, 1]$, such that the probabilities sum to 1. A discrete distribution is completely characterized by its probability mass function.

$$\text{(Finite discrete)} \quad \mathcal{V}[\![\texttt{Dist}\ \alpha]\!] \triangleq \mathcal{V}[\![\alpha]\!] \to [0,1] \quad \text{where } |\mathcal{V}[\![\alpha]\!]| \text{ finite}$$

As others have observed, probabilities form a monad (*e.g.*, [8, 25]). Monadic bind $x \leftarrow e_1\ ;\ e_2$ re-weights the probability mass function of e_2 by summing over all possible values e_1 can take on and re-weighting according to its probability. We write the expression denotation function as $\mathcal{E}[\![\Gamma \vdash e : \alpha]\!]\rho \in \mathcal{V}[\![\alpha]\!]$ under a well-typed environment ρ (w.r.t. Γ). Below, let $f_1 = \mathcal{E}[\![\Gamma \vdash e_1 : \texttt{Dist}\ \alpha_1]\!]\rho$ be the denotation of e_1.

$$\text{(Denotational)} \quad \mathcal{E}[\![\Gamma \vdash x \leftarrow e_1\ ;\ e_2 : \texttt{Dist}\ \alpha_2]\!]\rho\ (v_2) \triangleq \sum_{v_1 \in \mathrm{dom}(f_1)} \mathcal{E}[\![\Gamma, x : \alpha_1 \vdash e_2 : \texttt{Dist}\ \alpha_2]\!]\rho[x \mapsto v_1]\ (v_2) \cdot (f_1(v_1))$$

Alternatively, we can interpret monadic bind $x \leftarrow e_1\ ;\ e_2$ as sampling: "draw a sample according to distribution e_1, bind the value to variable x, and continue with distribution e_2." We can express sampling $\mathcal{S}[\![\Gamma \vdash e : \alpha]\!]\rho : 2^\omega \to \mathcal{V}[\![\alpha]\!]$ as bind in a state monad (written $\leftarrow_s$) whose state is a stream of bit-randomness 2^ω.

$$\text{(Sampling)} \quad \mathcal{S}[\![x \leftarrow e_1\ ;\ e_2]\!]\rho \triangleq v \leftarrow_s \mathcal{S}[\![e_1]\!]\rho\ ;\ \mathcal{S}[\![e_2]\!]\rho[x \mapsto v]$$

If we restrict the probabilities to be rational, then everything is discrete and finite, which coincides with traditional notions of computation. We can relate sampling to the denotational view as

$$\mathbf{P}(\mathcal{S}[\![e]\!]\rho = v) = \mathcal{E}[\![\Gamma \vdash e : \alpha]\!]\rho\ (v)$$

where the probability $\mathbf{P}(\cdot)$ is with respect to the distribution on the input bit-randomness.

Next, we can consider *continuous* distributions in the context of a probabilistic language. One approach is to interpret $\texttt{Dist}\ \alpha$ as a measure-theoretic distribution (*e.g.*, [3]). A *measure* $\mu : \mathcal{F} \to [0, \infty]$ on X maps a (measurable) set $A \in \mathcal{F}$, where $\mathcal{F}$ is a certain collection of subsets of X called a σ-algebra, to a non-negative real number, such that μ is countably additive and $\mu(\emptyset) = 0$. Then, a *probability measure* or *probability distribution* is a measure μ such that the mass of the entire space is 1 (*i.e.*, $\mu(X) = 1$).

$$\text{(Measure)} \quad \mathcal{V}[\![\texttt{Dist}\ \alpha]\!] \triangleq \mathcal{F} \to [0, 1] \quad \text{where } \mathcal{F} \text{ is a } \sigma\text{-algebra}$$

We can use monadic bind again, but now the re-weighting is accomplished by a (Lebesgue) integral.

$$\text{(Denotational)} \quad \mathcal{E}[\![\Gamma \vdash x \leftarrow e_1\ ;\ e_2 : \texttt{Dist}\ \alpha_2]\!]\rho(A) \triangleq$$
$$\int (\lambda v.\ \mathcal{E}[\![\Gamma, x : \alpha_1 \vdash e_2 : \texttt{Dist}\ \alpha_2]\!]\rho[x \mapsto v](A)) d(\mathcal{E}[\![\Gamma \vdash e_1 : \texttt{Dist}\ \alpha_1]\!]\rho)$$

Sampling has the same form as before. We can relate the sampling with the denotational view as the push-forward of μ_{iid} (distribution on 2^ω corresponding to independent and identically distributed (i.i.d.) fair coin flips) with respect to the sampling function.

$$\mu_{\text{iid}} \circ \mathcal{S}[\![e]\!]\rho^{-1} = \mathcal{E}[\![e]\!]\rho$$

Note that we could have also used the push-forward to relate the denotational view for the discrete case. Unlike the finite discrete case, we cannot implement $\mathcal{S}[\![\cdot]\!]\rho$ for all continuous distributions on a Turing machine. There are a countable number of Turing machine configurations, but an uncountable number of continuous distributions.

In this paper, we propose to address the gap between a denotational semantics with continuous distributions and an algorithmic sampling interpretation using Type-2 computable distributions. To this end, we will present a core probabilistic language extending PCF with distributions and give it semantics using largely standard techniques. Because we want the denotational semantics to be easily relatable to a sampling semantics, we will be restricted to considering only distributions on topological spaces (in this paper, computable metric spaces) instead of measurable spaces (*i.e.*, the tuple $(X, \mathcal{F})$ of a set and a σ-algebra) used in standard measure-theoretic probability. Thus, our semantics can support only Borel measures, *i.e.*, a measure defined uniquely on the open sets of the topology. Nonetheless, this covers a wide class of distributions used in practice, including familiar continuous distributions on the reals $\mathbb{R}$ (*e.g.*, Uniform, Gaussian, and etc.) and (infinite) products of reals.

After we give the semantics, we will identify a subset of the denotations as corresponding to Type-2 computable objects. Importantly, we can implement Type-2 computable operators in a standard programming language. Thus, we will provide an implementation of the sampling semantics as a Haskell library.

Moreover, the semantics of conditioning (when computable) can be given as a program that computes the conditional distribution (see Sec. 5). Hence, we can also give semantics to conditioning as a library function. We will see that we can write familiar probabilistic programs with familiar reasoning principles. Because we do not need anything beyond standard language semantics, our approach can also be used to give semantics to probabilistic languages embedded in full-fledged languages proposed and used by the machine learning community (*e.g.*, [9, 23, 35]).

3 Background on computability and distributions

In this section, we introduce background on Type-2 computability (see Type Two Theory of Effectivity or TTE [32]), which can be used to provide a notion of computability on reals and distributions. The background will primarily be useful for connecting the denotational semantics to an implementation of the sampling semantics (see Sec. 4.2). We start by illustrating the basic idea behind Type-2 computability using computable reals.

Intuitively, a real is computable if we can *enumerate* its binary expansion. Of course, a Turing machine algorithm can only enumerate a finite prefix of the expansion in a finite amount of time. Thus, Type-2 computability extends the conventional notion of algorithm (which terminates) to account for computing on bit-streams. A *Type-2 algorithm* is specified using conventional Turing machine code (*i.e.*, with finite set of states and finite state transition specification) and computes (partial) bit-stream functions (*i.e.*, $g_M : \{0,1\}^\omega \rightharpoonup \{0,1\}^\omega$) such that finite prefixes of the output are determined by finite prefixes of the input (called the *finite prefix property.*) This captures the intuition that a Type-2 algorithm computes a function to arbitrary precision in finite time using a finite amount of input, even though computing the entire output bit-stream cannot be done in finite time. Now that we have clarified what we mean by enumerate, we can return to computable reals.

More formally, a real $x \in \mathbb{R}$ is *computable* if we can enumerate a fast Cauchy sequence of rationals that converges to x. Recall that a sequence $(q_n)_{n\in\mathbb{N}}$ is *Cauchy* if for every $\epsilon > 0$, there is an N such that $d(q_n, q_m) < \epsilon$ 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 $d(q_n, q_{n+1}) < 2^{-n}$ for all n, we call $(q_n)_{n\in\mathbb{N}}$ a *fast Cauchy sequence*. Hence, the representation of a computable real as a fast Cauchy sequence evokes the idea of enumerating its binary expansion.

As an example of a computable real, consider π and one possible series expansion of it below [2].

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left[\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right]$$

An algorithm can use the above series expansion and a rate of convergence to obtain a fast Cauchy sequence (*e.g.*, the BPP algorithm [2]).

A function $f : \mathbb{R} \to \mathbb{R}$ is *computable* if given a fast Cauchy encoding of $x \in \mathbb{R}$, there is an algorithm that outputs a fast Cauchy sequence of $f(x)$. For example, addition $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is computable because an algorithm can add the (Cauchy) input sequences element-wise to obtain a (Cauchy) output sequence. Next, we introduce computable metric spaces, which can be used to generalize the ideas above to computable distributions.

As a reminder, *metric space* (X, d) is a set X equipped with a *metric* $d : X \times X \to \mathbb{R}$. 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.*, $B(c, r) = \{x \in X \mid d(c, x) < r\}$. In this paper, the topology we associate with a metric space will be the one induced by the collection of balls. Hence, a Borel measure will assign probabilities to open balls. For example, $(\mathbb{R}, d_{\text{Euclid}})$ gives the familiar Euclidean topology on the reals $\mathbb{R}$ (with Euclidean distance d_{Euclid}) and a Borel distribution on the reals assigns probabilities to open intervals.

To define a computable metric space, we need some additional properties on (X, d). First, we must be able to approximate elements of X using elements from a simpler, countable set S. This way, we can encode an element $s \in S$ as a finite sequence of bits and an element $x \in X$ as the stream of bits corresponding to a sequence of elements of S that converges to x. More formally, 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, (X, d) should be *complete*, *i.e.*, every Cauchy sequence comprised of elements from X also converges to a point in X. Putting this together gives the definition of a computable metric space.

Definition 1. *[12, Def. 2.4.1] A* computable metric space *is a tuple* (X, d, S) *such that*

- (X, d) *is a complete metric space.*
- S *is a countable, dense subset of* X *with a fixed numbering.*
- *For all* $i, j \in \mathbb{N}$, $d(s_i, s_j)$ *is computable, uniformly in* $\langle i, j \rangle$ *(*i.e.*, the function* $(i, j) \mapsto d(s_i, s_j)$ *is computable), where* $\langle \cdot, \cdot \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ *is a pairing function.*

The computable reals we gave at the beginning corresponds to the computable metric space $(\mathbb{R}, d_{\text{Euclid}}, \mathbb{Q})$.

A computable distribution over the computable metric space (X, d, S) can be defined as a computable point in the computable metric space $(\mathcal{M}(X), d_\rho, \mathcal{D}(S))$, 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 [12, Defn. 4.1.1.]), and $\mathcal{D}(S)$ is the class of distributions with finite support at ideal points S and rational masses (see [12, Prop. 4.1.1]). For instance, the sequence below converges to the standard Uniform distribution on the interval $(0, 1)$.

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

Thus, a Uniform distribution can be seen as the limit of a sequence of increasingly finer discrete, Uniform distributions. Although the idea of a computable

distribution as a computable point is fairly intuitive for the standard Uniform distribution, it may be less insightful for more complicated distributions.

Alternatively, we can think of a computable distribution on a computable metric space (X, d, S) in terms of sampling, *i.e.*, as a (Type-2) computable function $\{0,1\}^\omega \to X$. To make this more concrete, we sketch an algorithm that samples from the standard Uniform. The idea is to generate a value that can be queried for more precision instead of a sample x in its entirety. Thus, 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 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`, which recursively bisects an interval n times, starting with $(0, 1)$, using the random bit-stream u to select which interval to recurse on.

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

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

The sampling view is equivalent to the definition of a computable distribution in terms of computable metric spaces. From a practical perspective, this means we can implement probabilistic programs as samplers and still capture the class of computable distributions. For completeness, we state the equivalence below. First, a *computable probability space* [12, Def. 5.0.1] $(\mathcal{X}, \mu)$ is a pair where $\mathcal{X}$ is a computable metric space and μ is a computable distribution. A function $s : (\mathcal{X}, \mu) \to (\mathcal{Y}, \nu)$ is *measure-preserving* if $\nu(A) = \mu(s^{-1}(A))$ for all measurable A. We call a distribution μ on $\mathcal{X}$ *samplable* if there is a computable function $s : (2^\omega, \mu_{\text{iid}}) \to (\mathcal{X}, \mu)$ such that s is computable on $\text{dom}(s)$ of full-measure and is measure-preserving.

Proposition 1. *(Computable iff samplable) A distribution $\mu \in \mathcal{M}(X)$ on computable metric space $(X, d, \mathcal{S})$ is computable iff it is samplable (see [4, Lem. 2 and 3]).*

4 Semantics

In this section, we describe a core probabilistic language based on PCF and give it semantics using largely standard constructs (see [34]). The semantics of distributions will be given in terms of both valuations (a topological variant of distributions) and samplers. After we give the semantics, we identify a subset of the denotations as Type-2 computable, and give an implementation of the sampling semantics as a Haskell library. We will refer to Haskell plus the library as λ_{CD}.

4.1 A core language and its semantics

Syntax The core language extends a basic functional language (PCF with pairs) with reals (constants c) and distributions (constants d). The core language does not have a primitive for conditioning on a distribution. Instead, we will add conditioning as a library function later (see Sec. 5).

$$\begin{aligned}\alpha &::= \text{PCF types + pairs} \mid \texttt{Real} \mid \texttt{Samp}\ \alpha \\ e &::= \text{PCF exp + pairs} \mid c \mid d \mid e \oplus e \mid \texttt{return}\ e \mid x \leftarrow e\ ;\ e\end{aligned}$$

The primitives $\oplus$ provide primitive operations on reals. The expressions $\texttt{return}\ e$ and $x \leftarrow e_1\ ;\ e_2$ can be thought of as return and bind in the sampling monad. The type $\texttt{Real}$ refers to reals and the type $\texttt{Samp}\ \alpha$ refers to distributions. The typing rules are standard. The type $\texttt{Samp}\ \alpha$ is well-formed if values of type α support the operations required of a computable metric space. In this language, this includes naturals $\mathbb{N}$, reals $\mathbb{R}$, and products of computable metric spaces.

Interpretation of types The interpretation of types will use both complete partial orders (CPO's) and computable metric spaces. The former is a standard structure from denotational semantics used to give meaning to recursion. The latter was introduced in the background and are the spaces that we consider distributions on.

To start, we introduce basic notation for constructing CPO's that will be used to give the interpretation of types $\mathcal{V}[\![\cdot]\!]$. The construction $\mathrm{Disc}(X)$ equips the set X with the discrete order. $D_\perp$ creates a lifted domain with underlying set $\{\lfloor d \rfloor \mid d \in D\} \cup \{\perp\}$ with the usual lifted ordering. The CPO construction $D \Rightarrow E$ creates the CPO of continuous functions between CPO's D and E.

The interpretation of types $\mathcal{V}[\![\cdot]\!]$ is defined by induction on types. We give the interpretation of basic types (sans distributions $\texttt{Samp}\ \alpha$) below following a call-by-name evaluation strategy (to better match the Haskell implementation.)

$$\begin{aligned}\mathcal{V}[\![\texttt{Nat}]\!] &\triangleq \mathrm{Disc}(\mathbb{N})_\perp \\ \mathcal{V}[\![(\alpha_1, \alpha_2)]\!] &\triangleq (\mathcal{V}[\![\alpha_1]\!] \times \mathcal{V}[\![\alpha_2]\!])_\perp \\ \mathcal{V}[\![\alpha_1 \to \alpha_2]\!] &\triangleq (\mathcal{V}[\![\alpha_1]\!] \Rightarrow \mathcal{V}[\![\alpha_2]\!])_\perp \\ \mathcal{V}[\![\texttt{Real}]\!] &\triangleq \mathrm{Disc}(\mathbb{R})_\perp\end{aligned}$$

The interpretation of basic PCF types is standard. The reals $\mathbb{R}$ are given the discrete order (information ordering). In order to give the interpretation of $\texttt{Samp}\ \alpha$, we will need to write the space of distributions on $\mathcal{V}[\![\alpha]\!]$ as a CPO. To this end, we will use valuations, a topological variation of a distribution.

A *valuation* $\nu : \mathcal{O}(X) \to [0,1]$ is a function that assigns to each open set of a topological space $(X, \mathcal{O}(X))$ a probability, such that it is strict ($\nu(\emptyset) = 0$), monotone, and modular ($\nu(U) + \nu(V) = \nu(U \cup V) + \nu(U \cap V)$ for every open U and V). The type of a valuation can be given as a CPO. To see this, let $\mathcal{O}^{\subseteq}(X)$ be the open sets on a space X ordered by set inclusion and let $[0,1]^\uparrow$ be the interval $[0,1]$ ordered by $\leq$. Then, a valuation can be seen as an element of the CPO $\mathcal{O}^{\subseteq}(X) \Rightarrow [0,1]^\uparrow$. A valuation is similar to a measure (hence,

topological variant of distribution), but a valuation does not necessarily satisfy countable additivity. A valuation that does is called a *ω-continuous valuation*, where ω-continuous means $\nu(\cup_{n\in\mathbb{N}} O_n) = \sup_{n\in\mathbb{N}} \nu(O_n)$ for O_n open for $n \in \mathbb{N}$. Indeed, every Borel measure μ on $\mathcal{X}$ can be restricted to a *ω-continuous valuation* $\mu|_{\mathcal{O}(X)} : [\mathcal{O}^{\subseteq}(X) \Rightarrow [0,1]^{\uparrow}]$. Moreover, a computable distribution can be identified with a ω-continuous valuation (see [12, Prop. 4.2.1] and [28]). Now, we can proceed to interpret $\texttt{Samp}\ \alpha$.

One idea used in the study of probabilistic powerdomains is to use valuations to put distributions on CPO's (see [13]) using the CPO's Scott topology. For example, a valuation on the CPO Disc($\mathbb{N}$) results in the powerset $2^{\mathbb{N}}$, which is the σ-algebra associated with distributions on the naturals. However, we cannot apply a valuation in this manner to Disc($\mathbb{R}$) to obtain a familiar continuous distribution used in statistics. For example, a valuation on the Scott topology derived from the CPO Disc($\mathbb{R}$) results in the powerset $2^{\mathbb{R}}$, which is not the Borel σ-algebra on $\mathbb{R}$. Instead, we will start with the topology of a computable metric spaces which includes familiar continuous distributions, and then derive a CPO from it to support recursion.

To this end, we will use the *specialization preorder* (written S), which orders $x \sqsubseteq y$ if every open set that contains x also contains y, to convert a topological space $(X, \mathcal{O}(X))$ into a preordered set. Intuitively, $x \sqsubseteq y$ if x contains less information than y. We can always find an open ball that separates two distinct points x and y (the distance between two distinct points is positive) in a computable metric space. Hence, the specialization preorder of a computable metric space always gives the discrete order (information ordering), and hence degenerately, a CPO. For example, the specialization preorder of the computable metric space $(\mathbb{R}, d, \mathbb{Q})$ is Disc($\mathbb{R}$). We can now start to put this together to convert a computable metric space into a CPO.

We write $\mathcal{V}^M[\![\cdot]\!]$ to associate a well-formed type with a computable metric space, defined by induction on well-formed types.

$$\begin{aligned}
\mathcal{V}^M[\![\texttt{Nat}]\!] &\triangleq (\mathbb{N}, d_{\text{discrete}}, \mathbb{N}) \\
\mathcal{V}^M[\![\texttt{Real}]\!] &\triangleq (\mathbb{R}, d_{\text{Euclid}}, \mathbb{Q}) \\
\mathcal{V}^M[\![(\alpha_1, \alpha_2)]\!] &\triangleq \mathcal{V}^M[\![\alpha_1]\!] \times \mathcal{V}^M[\![\alpha_2]\!]
\end{aligned}$$

The interpretation of naturals and reals are standard. The interpretation of a product $\mathcal{V}^M[\![(\alpha_1, \alpha_2)]\!]$ forms the product of computable metric spaces $\mathcal{V}^M[\![\alpha_1]\!]$ and $\mathcal{V}^M[\![\alpha_2]\!]$. As one last step, we will need to handle $\bot$ for recursion. One can check the specialization preorder of the topology $\mathcal{O}(\lfloor X \rfloor) \cup \{\lfloor X \rfloor \cup \{\bot\}\}$ produces $\text{Disc}(X)_{\bot}$. We will write L to lift a computable metric space.

The interpretation of the type $\texttt{Samp}\ \alpha$ is a pair of a valuation and a sampling function realizing the valuation, where $\texttt{psh}_{\alpha}$ computes the push-forward measure and relates the valuation component to the sampling component. Below, let $D \times_F E$ be the CPO $\{(d, e) \in D \times E \mid F(e) = d\}$ with a product ordering, where F is a continuous function, and 2^{ω} is the CPO of continuous functions between

Disc($\mathbb{N}$) and Disc($\{0,1\}$).

$$\mathcal{V}[\![\texttt{Samp}\ \alpha]\!] \triangleq [\mathcal{O}^{\subseteq}(S \circ L(\mathcal{V}^M[\![\alpha]\!])) \Rightarrow [0,1]^{\uparrow}] \times_{\mathtt{psh}_\alpha} (2^\omega \Rightarrow S \circ L(\mathcal{V}^M[\![\alpha]\!])_\perp)\ .$$

This explicitly relates the valuation component to the sampling function component. We will see the implementation of the sampling semantics will identify $2^\omega \Rightarrow S \circ L(\mathcal{V}^M[\![\alpha]\!])_\perp$ with a Type-2 computable sampling algorithm.

The denotation of the sampler contains an extra lifting to distinguish `bot` : `Samp` α from `return bot` : `Samp` α, where $\mathcal{E}[\![\texttt{bot}]\!]\rho = \perp$. In the former, we obtain the bottom valuation, which assigns 0 mass to every open set. This corresponds to the sampling function $\lambda u \in 2^\omega.\ \perp$. In the latter, we obtain the valuation that assigns 0 mass to every open set, except for $\{\lfloor X \rfloor \cup \perp\}$ which is assigned mass 1. This corresponds to the sampling function $\lambda u \in 2^\omega.\ \lfloor \perp \rfloor$.

Semantics The expression denotation function $\mathcal{E}[\![\Gamma \vdash e : \alpha]\!]\rho \in \mathcal{V}[\![\alpha]\!]$ is defined by induction on the typing derivation. The denotation is also parameterized by a global environment Υ that interprets constants c and primitive distributions d. The following notation will be used for writing the semantics. We lift a function using $\dagger \in (D \Rightarrow E_\perp) \Rightarrow (D_\perp \Rightarrow E_\perp)$, defined in the usual way. The function $\mathrm{lift}(d) = \lfloor d \rfloor$ lifts an element. Finally, the notation let $x = e_1$ in e_2 is a strict let binding.

Now, we describe the expression denotation function. The semantics of the PCF expressions are standard and are not shown.

$$\begin{aligned}
\mathcal{E}[\![\Gamma \vdash c : \texttt{Real}]\!]\rho &\triangleq \Upsilon(c) \\
\mathcal{E}[\![\Gamma \vdash e_1 \oplus e_2 : \texttt{Real}]\!]\rho &\triangleq \mathcal{E}[\![e_1]\!]\rho \oplus_\perp \mathcal{E}[\![e_2]\!]\rho \\
\mathcal{E}[\![\Gamma \vdash d : \texttt{Samp}\ \alpha]\!]\rho &\triangleq \Upsilon(d) \\
\mathcal{E}[\![\Gamma \vdash \texttt{return}\ e : \texttt{Samp}\ \alpha]\!]\rho &\triangleq (\lambda U.\ 1_U(\mathcal{E}[\![e]\!]\rho), \lambda u.\ \lfloor \mathcal{E}[\![e]\!]\rho \rfloor) \\
\mathcal{E}[\![\Gamma \vdash x \leftarrow e_1\ ;\ e_2 : \texttt{Samp}\ \alpha_2]\!]\rho &= (\lambda U.\ \int h_U\ d\mu, g^\dagger \circ f) \quad \text{where} \\
\mu &= \pi_1(\mathcal{E}[\![e_1]\!]\rho) \\
h_U &= \lambda v.\ \pi_1(\mathcal{E}[\![e_2]\!]\rho[x \mapsto v])(U) \\
f &= \lambda u.\ \text{let } v = \pi_2(\mathcal{E}[\![e_1]\!]\rho)(u_e) \text{ in } \lfloor (v, u_o) \rfloor \\
g &= \lambda(v,w).\ \pi_2(\mathcal{E}[\![e_2]\!]\rho[x \mapsto v])(w)
\end{aligned}$$

The operations $\oplus_\perp$ correspond to strict primitives on reals. The denotation of `return` e is a point mass valuation centered at $\mathcal{E}[\![e]\!]\rho$, which corresponds to a sampler that ignores the input bit-randomness and returns $\mathcal{E}[\![e]\!]\rho$. The meaning of $x \leftarrow e_1\ ;\ e_2$ also gives a valuation and a sampler. In the former, we reweigh $\mathcal{E}[\![e_2]\!]\rho[x \mapsto \cdot]$ according to the valuation $\mathcal{E}[\![e_1]\!]\rho$. In the sampling view, we first split the input bit-randomness u into two bit streams u_e and u_o, corresponding to the bits of u with even indices and the bits of u with odd indices respectively. We run the sampler denoted by e_1 on the input u_e to produce a sample v, and pass the unused bit-randomness u_o. Then, we run the sampler $\pi_2(\mathcal{E}[\![e_2]\!]\rho[x \mapsto v])$ with x bound to v on the bit-randomness u_o.

Properties We need to check that the semantics we gave above is well-defined, particularly for the cases corresponding to manipulating distributions. Recall that the interpretation of $\texttt{Samp}\ \alpha$ is a pair of a valuation ν and a sampling function s, such that the push-forward of s is equivalent to ν. Thus, for the meaning of $\texttt{return}\ e$, we will need to argue that a sampler that ignores its input randomness is equivalent to a point mass valuation. For the meaning of $x \leftarrow e_1\ ;\ e_2$, we will need to relate a composition of sampling functions to a reweighing by integration. The first sampling function passes input bits that are unconsumed to the second sampling function. We can state this formally below.

Lemma 1. *(Push) Let X and Y be computable metric spaces.*

1. *Let $f \in 2^\omega \Rightarrow (S \circ L(X))_\perp$ be a constant function,* i.e.*, $f(u) = \lfloor d \rfloor$ for every u. Then, $\mathbf{psh}(f) = \lambda U.\ 1_U(d)$*
2. *Let $f \in 2^\omega \Rightarrow (S \circ L(X) \times 2^\omega)_\perp$ such that $\mathbf{psh}(f) = \mathbf{psh}((\mathit{lift} \circ \pi_1)^\dagger \circ f) \otimes \mu_{iid}$ (independence) and $g \in S \circ L(X) \times 2^\omega \Rightarrow (S \circ L(Y))_\perp$. Then,*

$$\mathbf{psh}(g^\dagger \circ f) = \lambda U.\ \int \lambda v.\ \mathbf{psh}(g(v))(U)\ d(\mathbf{psh}((\mathit{lift} \circ \pi_1)^\dagger \circ f))$$

Consequently, the expression denotation function is well-defined.

Lemma 2. *(Denotation well-defined) If $\Gamma \vdash e : \alpha$, then $\mathcal{E}[\![e]\!]\rho \in \mathcal{V}[\![\alpha]\!]$.*

We now restrict the interpretation of types to consider Type-2 computability. Importantly, we can implement the restriction in a standard programming language. Below, we give the interpretation of types $\mathcal{V}^c[\![\cdot]\!]$ restricted to Type-2 computable objects.

$$\begin{aligned}\mathcal{V}^c[\![\texttt{Real}]\!] &\triangleq \{r \in \mathcal{V}[\![\texttt{Real}]\!] \mid r \text{ is Type-2 computable or } r = \perp\}\\ \mathcal{V}^c[\![\texttt{Samp}\ \alpha]\!] &\triangleq \{(\nu, s) \in \mathcal{V}[\![\texttt{Samp}\ \alpha]\!] \mid \text{for Type-2 computable } u,\\ &\qquad s(u) \text{ is Type-2 computable on } \mathrm{dom}(s) \text{ or } s(u) = \perp\}\end{aligned}$$

We restrict the reals to the computable reals by choosing the Type-2 computable subset. The distributions are restricted to those sampling functions $2^\omega \Rightarrow (S \circ L(\mathcal{V}^M[\![\alpha]\!]))_\perp$ that are Type-2 computable. As a reminder, a Type-2 computable function $f : 2^\omega \to L(\mathcal{V}^M[\![\alpha]\!])$ transforms finite prefixes of an input in 2^ω into a fast Cauchy sequence in $\mathcal{V}^M[\![\alpha]\!]$. Hence, there are (continuous CPO) sampling functions in the unrestricted denotational semantics that do not satisfy this property.

The expression denotation function is still well-defined, assuming that our global environment only contains Type-2 computable elements and functions. Informally, it follows because Type-2 computable functions are closed under composition. For instance, the composition of Type-2 computable sampling algorithms will be Type-2 computable.

Lemma 3. *(Denotation well-defined) If $\Gamma \vdash e : \alpha$, then $\mathcal{E}[\![e]\!]\rho \in \mathcal{V}^c[\![\alpha]\!]$.*

```
module A (approx, anth, CMetrizable, enum, metric) where
approx :: (Nat -> a) -> A a     -- fast Cauchy sequence
anth :: A a -> Nat -> a

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

newtype A a = A { getA :: Nat -> a }

module CompDistLib (sampler) where
type RandBits = Nat -> Bool
type Samp = State RandBits
sampler :: (CMetrizable a) => (RandBits -> a) -> Samp a
```

Fig. 1. Library interface

In the next section, we will implement the sampling semantics as a Haskell library.

We end by giving familiar laws justified by the denotational semantics. For example, the type `Samp` α satisfies the monad laws and commutativity, where $\equiv$ means equivalence in distribution. This follows the monadic structure of standard probability as others have observed (*e.g.*, [8]). For commutativity, we require that x is not free in e_2 and that y is not free in e_1.

$$\begin{aligned}
x' \leftarrow \texttt{return}\ x\ ;\ f\ x' &\equiv f\ x && \text{(left identity)}\\
x \leftarrow e\ ;\ \texttt{return}\ x &\equiv e && \text{(right identity)}\\
x \leftarrow e\ ;\ y \leftarrow f\ x\ ;\ g\ y &\equiv y \leftarrow (x \leftarrow e\ ;\ f\ x)\ ;\ g\ y && \text{(bind associativity)}\\
x \leftarrow e_1\ ;\ y \leftarrow e_2\ ;\ f\ x\ y &\equiv y \leftarrow e_2\ ;\ x \leftarrow e_1\ ;\ f\ x\ y && \text{(commutativity)}
\end{aligned}$$

These laws have an operational sampling interpretation. For example, the associativity of bind says that an algorithm can re-associate the sampling steps, provided that there are no dependencies, and still obtain samples from the same distribution. Commutativity says that if two distributions are independent, then a sampler can sample them in either order. Of course, re-associating or commuting produces samplers that consume input randomness differently, but the distribution induced by the samplers will be equivalent.

4.2 Computable distributions as a library

We now present a Haskell library (Fig. 1) for expressing samplers that implements the ideas from the previous section. In particular, we will show that we can implement the sampling semantics in Haskell without assuming reals or primitive continuous distributions.

The module `A` contains operations for computable metric spaces. First, the type `A` α models an element of a computable metric space. It can be read as an

approximation by a sequence of values of type α. For example, a computable real can be given the type `CReal` $\triangleq$ `A Rat`, meaning it is a sequence of rationals that converges to a real. We can form values of type `A` α using `approx`, which requires us to check that the function we are coercing describes a fast Cauchy sequence, and project out approximations using `anth`.

To form `A` α, values of type α 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 -> A Rat
```

When we implement an instance of `CMetrizable`, we should check that the implementation of `enum` enumerates a dense subset and `metric` computes a metric as a computable metric space requires (see Sec. 3). Below, we give an instance of `A Rat` for computable reals.

```
instance CMetrizable Rat where
    enum = 0 : [ toRational m / 2^n
               | n <- [1..]
               , 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 (powers of 2), which are a dense subset of the reals. Note that there are many other choices here for the dense enumeration.[4] 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 `A` to implement computable operations on commonly used types. This reifies the computable primitives $\oplus$ from the core language as a library function. For example, a library for computable reals will contain the `CMetrizable` instance implementation above and other computable functions. However, some operations are not realizable (*e.g.*, equality) and so this module does not contain all operations one may want to perform on reals.

```
module RealLib (CReal, pi, ...) where
import A
import CompDistLib

type CReal = A Rat
instance CMetrizable Rat where ...

pi :: A Rat
(+) :: A Rat -> A Rat -> A Rat
exp :: A Rat -> A Rat
...
```

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

Next, we give constructs for expressing distributions described in the module `CompDistLib`. The type of sampling algorithms `Samp` α is an instance of the state monad.

```
type RandBits = Nat -> Bool
type Samp = State RandBits
```

The threaded state is an infinite bit-stream of randomness, where each bit is i.i.d. according to a Bernoulli distribution (*i.e.*, fair coin flip). An algorithm consumes this bit-stream to generate samples.

By reusing the state monad, we automatically obtain monadic bind and return from the Haskell standard library. Bind corresponds to sampling, where the sampling monad `Samp` α threads the bit-stream of randomness. Return corresponds to a deterministic computation because the computation ignores the bit-stream. Instead of building in primitive distributions, we provide an introduction form `sampler` that coerces an arbitrary sampling function, constrained to types α that have a `CMetrizable` instance. We should call `sampler` only on sampling functions realizing Type-2 computable sampling algorithms.

```
sampler :: (CMetrizable a) => (RandBits -> a) -> Samp a
sampler f = State (\u -> let (u_e, u_o) = split u in
                         return (f u_e, u_o))
    where split u = (\n -> u (2 * n), \n -> u (2 * n + 1))
```

The function `split` splits an input bit-stream of randomness into two, non-overlapping bit-streams of randomness and will be used to ensure that primitive distributions built from `sampler` have access to fresh bit-streams of randomness. Thus, `sampler` first splits the input randomness `u`. Then, it runs the input sampling function f on `u_e`, and threads the unused portion `u_o` through for the rest of the computation. At this point, we are done describing the implementation of the semantics. We end with a few examples.

First, note that we can express distributions that are not (strictly) Type-2 computable—recall from the definition that computable distributions are normalized (see Sec. 3). Consider the program below, where `bot` = `bot` (*i.e.*, `bot` is a computation that does not terminate) and `bernoulli` is a Bernoulli distribution:

```
maybeLoop :: Samp Bool
maybeLoop = do
  b <- bernoulli
  if b then return bot else bernoulli
```

This program diverges 1/2 the time and returns a fair coin flip the other 1/2 according to Haskell semantics. Hence, it does not generate a sample with probability 1 so it cannot be a Type-2 computable distribution. Later when we add conditioning to λ_{CD} as a library function implementing a Type-2 conditioning algorithm, we will not give semantics to conditioning on un-normalized distributions because the Type-2 algorithm assumes the distribution is normalized. Distributions such as `maybeLoop` that fail to generate a sample with positive probability are not useful in the context of Bayesian inference. Hence, we are fine not giving semantics to conditioning on these distributions.

Next, we show how to use the library to encode continuous distributions. To start, we fill in the sketch of the standard Uniform distribution from before using `sampler`. As a reminder, we need to convert a random bit-stream into a sequence of (dyadic) rational approximations.

```
std_uni :: Samp CReal
std_uni = sampler (\u -> A (\n -> bisect (n+1) u 0 1 0))
    where
      bisect n u (l :: Rat) (r :: Rat) m =
          if m < n
          then (if u m
                then bisect n u l (midpt l r) (m+1)
                else bisect n u (midpt l r) r (m+1))
          else midpt l r
      midpt l r = l + (r - l) / 2
```

The function `bisect` repeatedly bisects an interval specified by $(\mathtt{l}, \mathtt{r})$. 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 Uniform distribution, we can encode other primitive distributions (*e.g.*, Normal, Exponential, and etc.) as transformations of the Uniform as in standard statistics using return and bind.

For example, we give an encoding of the standard Normal distribution using the Marsaglia polar transformation, which diverges with probability 0:

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

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 `std_uniform`. We can check that this distribution is samplable. First, we check that the algorithm produces a sample with probability 1 by showing that both $s = 1$ (by absolute continuity) and divergence (by Borel-Cantelli) occur with probability 0. Note that the operation `<` semi-decides both $<$ and $>$, where we are guaranteed with probability 1 that equality does not hold. Next, the algorithm is measure-preserving because `uniform` (-1) 1 is a samplable distribution and compositions of computable functions preserve measure. Hence, we can conclude that this distribution is samplable.

5 Conditioning

Conditioning is the core operation of Bayesian inference. As we alluded to earlier, conditioning is not computable in general [1]. We could use a more abstract

definition of conditioning used in measure theory, but it would be undesirable if the semantics of conditioning for a probabilistic programming language was not computable given that one of its goals is to automate inference. Instead, we take a library approach, which requires the client to provide an implementation of a conditioning algorithm and limits us to situations where conditioning is computable. Hence, the semantics of our core language remains unchanged.

5.1 Preliminaries

We begin with background on conditioning in the context of Bayesian inference before moving to conditioning in the measure-theoretic and computable settings. A Bayesian model puts a distribution on the product space $\Theta \times D$, where Θ is the space of parameters and D is the space of observations. Observing a particular value $d \in D$ restricts the domain of the distribution to that subspace, resulting in a distribution on parameters given the data d. More formally, the objective of Bayesian inference is to compute the *posterior distribution* $p(\theta \mid d)$ (function of θ for fixed d) given a *joint distribution* $p(d, \theta)$ and observed data $d \in D$. The notation $p(\cdot)$ refers to the density of a distribution, but it is common (in statistics) to refer to $p(\cdot)$ as a distribution as we have done above. The *density* of a distribution μ with respect to a distribution ν is an integrable function f such that $\mu(A) = \int_A f d\nu$ for every measurable A. Thus, a density along with the underlying measure ν (often Lebesgue measure) determines a measure. The joint and posterior are related with *Bayes' rule*:

$$p(\theta \mid d) = \frac{p(d, \theta)}{\int p(d, \theta) d\theta} \propto p(d \mid \theta) p(\theta) \ ,$$

where $p(\theta)$ is called the *prior distribution* and $p(d \mid \theta)$ is called the *likelihood*. The posterior distribution $p(\theta \mid d)$ is a *conditional distribution*.

Conditioning can be defined in more abstract settings when we do not have densities. First, we need to introduce additional measure-theoretic definitions. A tuple $(\Omega, \mathcal{F}, \mu)$ is called a *probability space*, where $(\Omega, \mathcal{F})$ is a measurable space and μ is a measure. We will omit the underlying σ-algebra and measure when they are unambiguous from context. A function $f : \Omega_1 \to \Omega_2$ is *measurable* if $f^{-1}(B)$ is measurable for measurable B. In the case of metric spaces, the measurable sets are generated by the open balls so measurable functions are a superset of the continuous functions. A *random variable* that takes on values in a probability space S is a measurable function $X : \Omega \to S$. The *distribution* of a random variable X, written $\mathbf{P}[X \in \cdot]$, is defined as the push-forward of the underlying measure, *i.e.*, $\mathbf{P}[X \in A] = \mu(X^{-1}(A))$ for all measurable A. We write $X \sim \mu$ to indicate that the random variable X is distributed according to μ.

Now, we can give definitions for conditioning in the measure-theoretic setting. We will not give the most general definition of conditioning as conditional expectation, choosing to restrict the scope to the case that also applies in the computable setting [1]. In the following, let X and Y be random variables in computable metric spaces S and T respectively. In addition, let $\mathcal{B}_T$ be the (Borel)

```
module Conditioning where
import A
import CompDistLib
import RealLib

newtype BndDens a b =
    BndDens { getDens :: (A a -> A b -> CReal, Rat) }

-- Requires comp. dist. and bounded conditional density
obs_dens :: forall u v y.
    (CMetrizable u, CMetrizable v, CMetrizable y) =>
    Samp (A (u, v)) -> BndDens u y -> A y -> Samp (A (u, v))
```

Fig. 2. An interface for conditioning.

σ-algebra on T and let $\mathbf{P}_X$ be the distribution of X. A measurable function $\mathbf{P}[Y \in B \mid X]$ for B measurable is a *version* of the *conditional probability of* $Y \in B$ *given* X when

$$\mathbf{P}(X \in A, Y \in B) = \int_A \mathbf{P}[Y \in B \mid X] d\mathbf{P}_X$$

for all measurable A. A *probability kernel* is a function $\kappa : S \times \mathcal{B}_T \to [0,1]$ such that $\kappa(s, \cdot)$ is a Borel measure on T for every $s \in S$, and $\kappa(\cdot, B)$ is measurable for every measurable B. A *regular conditional probability* is then a probability kernel κ such that $\mathbf{P}(X \in A, Y \in B) = \int \kappa(x, B) \mathbf{P}_X(dx)$, where A and B are measurable.

These definitions have computable versions.

Definition 2. *(Computable probability kernel [1, Def. 4.2]) Let S and T be computable metric spaces, and $\mathcal{B}_T$ be the σ-algebra on T. A probability kernel $\kappa : S \times \mathcal{B}_T \to [0,1]$ is computable if $\kappa(\cdot, A)$ is a lower semi-computable function for every r.e. open $A \in \sigma(\mathcal{B}_T)$.*

Definition 3. *(Computable conditional distribution [1, Def. 4.7]) Let X and Y be random variables in computable metric spaces S and T. Let κ be a version of $\mathbf{P}[Y \mid X]$ (notation for $\mathbf{P}[Y \in \cdot \mid X]$). Then $\mathbf{P}[Y \mid X]$ is computable if κ is computable on a $\mathbf{P}_X$ measure-one subset.*

Thus, a non-computable conditional distribution is one for which *every* version is non-computable.

5.2 Conditioning

Now, we add conditioning as a library to λ_{CD} (Fig. 2). λ_{CD} provides only a restricted conditioning operation `obs_dens`, which requires a conditional density. We will see that the computability of `obs_dens` corresponds to an effective

version of Bayes' rule, which is central to Bayesian inference, and hence, widely applicable in practice. 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 [7, Prop. 3.1.2]).

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

Proposition 2. *[1, Cor. 8.8] Let U, V and Y be computable random variables, where Y is independent of V given U. Let $p_{Y \mid U}(y \mid u)$ be a conditional density of Y given U that is bounded and computable. Then the conditional distribution $\mathbf{P}[(U,V) \mid Y]$ is computable.*

The bounded and computable conditional density enables the following integral to be computed, which is in essence Bayes' rule. A version of the conditional distribution $\mathbf{P}((U,V) \mid Y)$ is

$$\kappa_{(U,V) \mid Y}(y, B) = \frac{\int_B p_{Y \mid U}(y \mid u) d\mathbf{P}_{(U,V)}}{\int p_{Y \mid U}(y \mid u) d\mathbf{P}_{(U,V)}}$$

where B is a Borel set in the space associated with $U \times V$.

Another interpretation of the restricted situation is that our observations have been corrupted by independent smooth noise [1, Cor. 8.9]. To see this, let U be the random variable corresponding to our ideal model of how the data was generated, V be the random variable corresponding to the model parameters, and Y be the random variable corresponding to the corrupted data we observe. Notice that the model (U,V) is not required to have a density and can be an arbitrary computable distribution. Indeed, probabilistic programming systems proposed by the machine learning community impose a similar restriction (*e.g.*, [9, 35]).

Now, we describe `obs_dens`, starting with its type signature. Let the type `BndDens` α β represent a bounded computable density:

```
newtype BndDens a b =
    BndDens { getDens :: (A a -> A b -> CReal, 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 `maybeLoop` that diverge with positive probability. Hence, we do not give semantics to conditioning on those distributions.

Now, we give a sketch of its implementation. In essence, it is 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.[5]

[5] That is, we implement the Type-2 machine code as a Haskell program. The implementation relies on Haskell's imprecise exceptions mechanism [22] to express

```
obs_dens :: forall u v y.
    (CMetrizable u, CMetrizable v, CMetrizable y) =>
    Samp (A (u, v)) -> BndDens u y -> A y -> Samp (A (u, v))
obs_dens dist (BndDens (dens, bnd)) d =
    let f :: A (u, v) -> CReal = \x -> dens (afst x) d
        mu :: Prob (u, v) = stc dist
        nu :: Prob (u, v) = \bs ->
              let num  = integrate_bnd_dom mu f bnd bs
                  denom = integrate_bnd mu f bnd
              in map fst (cauchy_to_lu (num / denom))
    in
      cts nu
```

The parameter `dist` corresponds to the joint distribution of the model (both model parameters and likelihood), `dens` corresponds to a bounded conditional density 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 functions `stc` and `cts` witness the computable isomorphism between samplable and computable distributions. The functions `integrate_bnd_dom` and `integrate_bnd` compute an integral (see [12, Prop. 4.3.1]), and correspond to an effective Lebesgue integral. `cauchy_to_lu` converts a Cauchy description of a computable real into an enumeration of lower and upper bounds.

Because `obs_dens` 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 by conditioning on equivalent probability zero events [26]. In addition, note that 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.

Now, we give an encoding in λ_{CD} of an example by Ackerman et al. [1] that shows that conditioning is not always computable. Similar to other results in computability theory, the example demonstrates that an algorithm computing the conditional distribution would also solve the Halting problem.

```
non_comp :: Samp (Nat, CReal)
non_comp = do
  n <- geometric (1/2)
  c <- bernoulli (1/3)
  u <- uniform 0 1
  v <- uniform 0 1
  x <- return (approx (\k -> dk k (tm_halts_within_k n k)))
  return (n, x)
    where dk k m | m > k = anth v k
                 | m = k = c
                 | m < k = anth u (k - m - 1)
```

---

the modulus of continuity of a computable function (see Andrej Bauer's blog http://math.andrej.com/2006/03/27/sometimes-all-functions-are-continuous).

The Uniform distribution `uniform` generates approximations via dyadic rationals. The distributions `geometric` and `bernoulli` correspond to Geometric and Bernoulli distributions. The function `tm_halts_within_k` accepts a natural n specifying the n-th Turing machine and a natural k describing the number of steps to run the machine for, and returns the number of steps the n-th Turing machine halts in or k if it cannot tell. Upon inspection, we see the function `dk` produces the binary expansion (as a dyadic rational) of a computable real, using `tm_halts_within_k` to select different bits of the binary expansion of u or v, or the bit c. Thus, it is computable. However, it is not possible to compute the conditional distribution $P(N \mid X)$, where the random variable N corresponds to the program variable n and X to x, because we would compute the complement of the Halting set.

6 Examples

In this section, we give additional examples of distributions in λ_{CD}, including a non-parametric prior (a distribution on a countable number of parameters) and a singular distribution (neither discrete nor continuous). This highlights the expressiveness of λ_{CD} and demonstrates how to apply the reasoning principles from before.

Geometric distribution Consider the encoding of a Geometric distribution with bias 1/2, which returns the number of Bernoulli trials until a success.

```
geometric :: SampT Nat
geometric = do
  b <- bernoulli (1/2)
  if b
  then return 1
  else do
    n <- geometric
    return (n + 1)
```

Our denotational view shows that the code encodes the Geometric distribution, where μ_B is a Bernoulli distribution and μ^n corresponds to n un-foldings of `geometric`. To see this, we can proceed by induction on n and use the induction hypothesis that μ^n is the measure $\{0\} \mapsto 0, \{1\} \mapsto (1/2), \ldots, \{n\} \mapsto (1/2)^n$.

$$\mathcal{E}[\![\texttt{geometric}]\!]\rho = \lambda U. \sup_n \int \lambda v. \begin{cases} 1_U(1) & \text{if } v = t \\ \int \lambda w.\, 1_U(w+1) d\mu^n & \text{otherwise} \end{cases} d\mu_B$$

$$= \lambda U. \sup_n (1_U(1)\frac{1}{2} + \sum_{w=0}^{\infty} 1_U(w+1)\mu^n(\{w\}))$$

Non-parametric prior We give two different encodings of the Dirichlet process, a prior distribution used in mixture models where the number of mixtures is unknown (*e.g.*, [19]). The Dirichlet process $\mathrm{DP}(\alpha, G_0)$ is a distribution on distributions—a draw produces a discrete distribution with support determined

by G_0, the based distribution, and mass according to α, the concentration parameter. The Dirichlet process can be represented in multiple ways, where each representation illuminates different properties. One representation is called the Blackwell-MacQueen urn scheme (see [19]), which describes how to sample from the distribution resulting from a draw of the Dirichlet process. Thus, we can imagine it describing the following process:

$$\begin{aligned} G &\sim \mathrm{DP}(\alpha, G_0) \\ \theta_n \mid G &\sim G \text{ for } n \in \mathbb{N} \,. \end{aligned}$$

The conditional distribution of θ_n is

$$\theta_n \mid \theta_{1:n-1} \sim \frac{\alpha}{\alpha+n-1} G_0 + \frac{1}{\alpha+n-1} \sum_{j=1}^{n-1} 1_{\theta_j} \text{ for } n \in \mathbb{N} \,,$$

which shows that the base distribution G_0 determines the support and α determines how often we select a new point from G_0 to put mass on. We can encode the conditional distribution in λ_{CD}.

```
urn' :: CReal -> Samp a -> [a] -> Samp a
urn' alpha g0 prev =
    let l = length prev
        n :: Integer = (toInteger l) + 1
        w = 1 / (fromInteger n - 1 + alpha)
        ws = replicate l w ++ [alpha]
        d = disc_id ws
    in do
      c <- d
      if c == n - 1
      then g0
      else return (prev !! (fromInteger c))
```

We can put our reasoning principles to work to argue that `urn'` encodes the conditional distributions. First, we can use a distributional view of the monadic block of `urn'` under an environment ρ, where $\mu = \mathcal{E}[\![\texttt{d}]\!]\rho$.

$$\mathcal{E}[\![\texttt{c} \leftarrow \texttt{d}\ ;\ \texttt{if c == n-1 then g0 else return (prev !! c)}]\!]\rho = \lambda U.$$
$$\int \lambda v. \pi_1 \circ \begin{cases} \mathcal{E}[\![\texttt{g0}]\!]\rho[c \mapsto v](U) & \text{if } \mathcal{E}[\![\texttt{c == n-1}]\!]\rho[c \mapsto v] \\ \mathcal{E}[\![\texttt{return (prev !! c)}]\!]\rho[c \mapsto v](U) & \text{otherwise} \end{cases} d\mu$$

Next, substituting away the `let` bindings (justified by Haskell semantics) implies that $\mathcal{E}[\![d]\!]\rho$ is the discrete distribution

$$0 \mapsto \frac{1}{\alpha+n-1}, \ldots, n-2 \mapsto \frac{1}{\alpha+n-1}, n-1 \mapsto \frac{\alpha}{\alpha+n-1} \,.$$

This reduces the previous integral to the summation

$$\lambda U.\ \sum_{j=0}^{n-2} \frac{1}{\alpha+n-1} \mathcal{E}[\![\texttt{return (prev !! c)}]\!]\rho[c \mapsto j](U) + \frac{\alpha}{\alpha+n-1} \mathcal{E}[\![\texttt{g0}]\!]\rho[c \mapsto n-1](U)\ ,$$

where $\mathcal{E}[\![\texttt{g0}]\!]\rho$ is the base distribution G_0. Rewriting this in statistical notation gives the desired result

$$\mathcal{E}[\![\texttt{urn' alpha g0 prev}]\!]\rho \sim \frac{\alpha}{\alpha+n-1} G_0 + \frac{\sum_{j=1}^{n-1}}{\alpha+n-1} 1_{\texttt{prev}_j}\ .$$

Next, we can describe the entire infinite sequence using lazy monadic lists

```
data MList m a = Nil | Cons a (m (MList m a))
```

and analogoulsy define common operations expected of lists such as `iterate`, `map`, and `tail`.

```
urn :: forall a. CReal -> Samp a -> Samp (MList a)
urn alpha g0 =
    let f :: ((a, [a]) -> Samp a) = return . fst)
        g (_, acc) = do
          x <- urn' alpha g0 acc
          return (x, acc ++ [x])
    in do
      x0 <- g0
      xs <- ML.map f (ML.iterate g (return (x0, [])))
      ML.tail xs
```

Expressing the resulting conditional distribution for each n gives

$$\mathcal{E}[\![\texttt{urn alpha g0}]\!]\rho_n \mid \mathcal{E}[\![\texttt{urn alpha g0}]\!]\rho_{1:n-1} \sim \frac{\alpha}{\alpha+n-1} G_0 + \frac{\sum_{j=1}^{n-1}}{\alpha+n-1} 1_{\texttt{psh}(\mathcal{E}[\![\texttt{urn alpha g0}]\!]\rho)_j}\ .$$

Alternatively, there is a constructive representation known as the stick-breaking construction (see [19]) that gives the structure of the discrete distribution directly. We describe a process that gives $G \sim \mathrm{DP}(\alpha, G_0)$. First, let random variables $\beta_k \sim \mathrm{Beta}(1, \alpha)$ be distributed according to the Beta distribution for $k \in \mathbb{N}$. Next, define $\pi_k = \beta_k \prod_{i=1}^{k-1}(1-\beta_i)$ for $k \in \mathbb{N}$. Let $X_k \sim G_0$ for $k \in \mathbb{N}$. The result G is $\sum_{k=1}^{\infty} \pi_k 1_{X_k}(\cdot)$. We can encode the stick-breaking construction in λ_{CD}, where `ML.@:` is synonymously `ML.Cons` and `ML.!!!` indexes a lazy monadic list. The function `mdisc_id` samples an index according to an input lazy monad list specifying probabilities.

```
sticks :: CReal -> Samp a -> Samp a
sticks alpha g0 = do
```

```
xs <- ML.repeat g0
pis <- weights 1
c <- mdisc_id pis
xs ML.!!! fromInteger c
  where
    weights :: CReal -> Samp (MList CReal)
    weights left = do
      v <- beta 1 alpha
      return ((v * left) ML.@: (weights (left * (1 - v))))
```

We can follow a similar pattern to reason about the urn representation. For instance, we can analyze this function compositionally as before by reasoning that $c \leftarrow$ `mdisc_id pis xs` ; `ML.!!! fromInteger c` selects a sample from `xs` according to the weights `pis`. Finally, we can combine this with showing that `weights` generates the weights π_k.

The encodings show that we can use probabilistic and standard program reasoning principles at the same time. Because we checked that each program encoded their respective representation, we also obtain that sampling `sticks` an infinite number of times is equivalent to `urn` because both encode the Dirichlet process. This might seem strange because the urn encoding has more sequential dependencides than the stick-breaking representation. The equivalence relies on a probabilistic concept called *exchangeability*, which asserts the existence of a conditionally independent representation if the distribution is invariant under all finite permutations. Exchangeability has been studied in the Type-2 setting [5] and it would be interesting to see if we can lift those results into λ_{CD}.

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. 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 in λ_{CD} be directly transforming a random bit-stream into a sequence of approximations.

```
cantor :: Samp CReal
cantor = sampler (\u -> approx (\n -> go u 0 1 0 n))
    where go u (left :: Rat) (right :: Rat) n m =
              let pow = 3 ^^ (-n) in
              if (n < m)
              then (if u n
                    then go u left (left + pow) (n + 1) m
                    else go u (right - pow) right (n + 1) m)
              else right - (1 / 2) * pow
```

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. Crucially, the number of trisections is bounded by the precision we would like to generate the sample to. We could express

the Cantor distribution in a measure-theoretic language with recursion, but we would need to trisect infinitely to express the distribution exactly.

7 Related Work

The semantics of probabilistic programs have been studied outside the context of Bayesian inference, in particular, to look at non-determinism and probabilistic algorithms. For instance, Kozen [15] gives both a sampling semantics and denotational semantics to a first-order imperative language. Instead of CPO constructions, the denotational semantics uses constructs from analysis, and recovers order-theoretic structure to support recursion. In the functional setting, researchers have studied *probabilistic powerdomains*, which put *distributions* on CPO's so that the space of distributions themselves forms a CPO. For example, Saheb [27] introduces the probabilistic powerdomain on CPO's by considering probability measures. Jones [13] considers powerdomains of valuations on CPO's instead and shows that this results in CPO's that have more desirable properties reminiscent of standard domains (*e.g.*, ω-algebraic). The work on powerdomains typically does not consider continuous distributions (we obtain the topology from a computable metric space as opposed to the Scott-topology of a CPO) or Bayesian inference.

Semantics for probabilistic programs have also been studied with machine learning applications in mind. Ramsey *et al.* give denotational semantics to a higher-order language without recursion extended with discrete distributions using the probability monad, and shows how to efficiently implement probabilistic queries [25]. Borgström *et al.* [3] uses measure theory to give denotational semantics to a first-order language without recursion using measure transformers. The main focus of the work is to ensure that the semantics of conditioning on events of 0 probability is well-defined. Toronto et al. [29] propose another measure-theoretic approach for a first-order language with recursion by interpreting probabilistic programs as pre-image computations of measurable functions to support conditioning. They do not use standard constructs from denotational semantics, and thus, do not handle recursion with a fixed-point construction. Instead, their target is a variant of lambda calculus extended with set-theoretic operations (λ_{ZFC}). Park *et al.* [21] give an operational semantics to an ML-like language in terms of sampling functions, but uses an idealized abstract machine. Hence, they also use the observation that a continuous distribution is characterized by a sampling procedure, but do not explore connections to a denotational approach nor the (faithful) realizability of their operational semantics.

Probabilistic languages have also been proposed to study inference, and typically have been given operational semantics. Some languages restrict to expressing well-established abstractions, such as factor graphs and Bayesian networks, in order to automate standard inference algorithms. In this restricted setting, they are given semantics in terms of the abstractions they express. For instance, the Bugs system [16] provides a language for users to specify Bayesian networks and implements Gibbs sampling, a Markov Chain Monte Carlo sampling algo-

rithm used in practice that reduces the problem of sampling from a multivariate conditional distribution into sampling from multiple univariate conditional distributions. Several other probabilistic programming languages similar to Bugs have been developed for expressing factor graphs (*e.g.*, [17, 18, 10]) and directed models (*e.g.*, [11, 30]), and automate inference using standard algorithms, including message passing and HMC sampling.

Other researchers extend Turing-complete languages with the ability to sample from primitive distributions (*e.g.*, [14, 24, 9, 33, 23, 35, 20]) and have been proposed to study inference in this richer setting. These languages have operational semantics given in terms of an inference method implemented in the host language. Examples include Stochastic Lisp [14] and Ibal [24] which only have discrete distributions. Others add *continuous* distributions, including Church [9] which embeds in Scheme and Fiagro [23] which embeds in Scala. The Church language performs inference by sampling program traces. Other languages have built upon this, including Probabilistic Matlab [33], Probabilistic C [35], and R2 [20], each proposing a different method to improve the efficiency of sampling program traces.

8 Discussion

In summary, we show that we can give sampling semantics to a high-level probabilistic language for Bayesian inference that corresponds to a natural denotational view. Our approach, by acknowledging the limits of computability, gives a semantics that corresponds to the intuition of probabilistic programs encoding generative models as samplers. In particular, Type-2 computability makes sense (*i.e.*, algorithmically) of sampling from continuous distributions as well as the difficulty of supporting a generic conditioning primitive. Moreover, we have shown that many ideas such as the ones in the probabilistic powerdomains can also be applied to give semantics to modern probabilistic languages designed for Bayesian inference. We end with a few directions for future work.

First, as we mentioned previously, algorithms that operate on computable distributions (and reals) are not necessarily efficient. In many practical situations, it is not necessary to compute to arbitrary precision as Type-2 computability demands, but *enough* precision. It would be interesting to see if we can efficiently implement an approximate semantics in this relaxed setting. Second, our approach has not been designed with inference in mind. The language λ_{CD} is perhaps too expressive—we can express distributions that are not meaningful for inference (*e.g.*, `maybeLoop`) and singular distributions that have limited applications (*e.g.*, Cantor distribution). It would be interesting to explore restricted language designs where we have guaranteed efficient inference.

Acknowledgements This work was supported by Oracle Labs. We would like to thank Nate Ackerman, Stephen Chong, Jean-Baptiste Tristan, Dexter Kozen, Dan Roy, and our anonymous reviewers for helpful discussions and feedback.

References

1. Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. Noncomputable Conditional Distributions. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, LICS '11, pages 107–116, Washington, DC, USA, 2011. IEEE Computer Society.
2. David Bailey, Peter Borwein, and Simon Plouffe. On the Rapid Computation of Various Polylogarithmic Constants. *Math. Comput.*, 66(218):903–913, April 1997.
3. Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure Transformer Semantics for Bayesian Machine Learning. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 77–96, Berlin, Heidelberg, 2011. Springer-Verlag.
4. Cameron E. Freer and Daniel M. Roy. Posterior distributions are computable from predictive distributions. In Yee W. Teh and D. M. Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS-10)*, volume 9, pages 233–240, 2010.
5. Cameron E Freer and Daniel M Roy. Computable de Finetti measures. *Annals of Pure and Applied Logic*, 163(5):530–546, 2012.
6. Peter Gács. Uniform test of algorithmic randomness over a general space. *Theoretical Computer Science*, 341(1):91–137, 2005.
7. Stefano Galatolo, Mathieu Hoyrup, and Cristóbal Rojas. Effective symbolic dynamics, random points, statistical behavior, complexity and entropy. *Information and Computation*, 208(1):23–41, 2010.
8. Michele Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.
9. N. D. Goodman, V. K. Mansinghka, D. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence, UAI 2008*, pages 220–229, 2008.
10. S. Hershey, J. Bernstein, B. Bradley, A. Schweitzer, N. Stein, T. Weber, and B. Vigoda. Accelerating Inference: towards a full Language, Compiler and Hardware stack. *CoRR*, abs/1212.2991, 2012.
11. Matthew D. Hoffman and Andrew Gelman. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, in press, 2013.
12. Mathieu Hoyrup and Cristóbal Rojas. Computability of probability measures and Martin-Löf randomness over metric spaces. *Information and Computation*, 207(7):830–847, 2009.
13. C. Jones and G. Plotkin. A Probabilistic Powerdomain of Evaluations. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 186–195, Piscataway, NJ, USA, 1989. IEEE Press.
14. D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian Inference for Stochastic Programs. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI)*, pages 740–747, 1997.
15. Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.
16. D. Lunn, D. Spiegelhalter, A. Thomas, and N. Best. The BUGS project: Evolution, critique and future directions. *Statistic in Medicine*, 2009.

17. A. McCallum, K. Schultz, and S. Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems 22*, pages 1249–1257, 2009.
18. T. Minka, J. Guiver J. Winn, and A. Kannan. Infer.NET 2.3, 2009. Microsoft Research Cambridge.
19. Radford M Neal. Markov chain sampling methods for Dirichlet process mixture models. *Journal of computational and graphical statistics*, 9(2):249–265, 2000.
20. Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. R2: An efficient MCMC sampler for probabilistic programs. In *AAAI Conference on Artificial Intelligence*, 2014.
21. Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 171–182, New York, NY, USA, 2005. ACM.
22. Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A Semantics for Imprecise Exceptions. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 25–36, New York, NY, USA, 1999. ACM.
23. A. Pfeffer. Creating and Manipulating Probabilistic Programs with Figaro. In *2nd International Workshop on Statistical Relational AI*, 2012.
24. Avi Pfeffer. IBAL: A Probabilistic Rational Programming Language. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'01, pages 733–740, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
25. Norman Ramsey and Avi Pfeffer. Stochastic Lambda Calculus and Monads of Probability Distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 154–165, New York, NY, USA, 2002. ACM.
26. M.M. Rao and R. J. Swift. *Probability Theory with Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
27. Nasser Saheb-Djahromi. CPO's of measures for nondeterminism. *Theoretical Computer Science*, 12(1):19–37, 1980.
28. Matthias Schröder. Admissible representations for probability measures. *Mathematical Logic Quarterly*, 53(4-5):431–445, 2007.
29. Neil Toronto, Jay McCarthy, and David Van Horn. Running probabilistic programs backwards. In *Programming Languages and Systems*, pages 53–79. Springer, 2015.
30. Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C. Pocock, Stephen Green, and Guy L. Steele. Augur: Data-Parallel Probabilistic Modeling. In *Advances in Neural Information Processing Systems*, pages 2600–2608, 2014.
31. Klaus Weihrauch. Computability on the probability measures on the Borel sets of the unit interval. *Theoretical Computer Science*, 219(1):421–437, 1999.
32. Klaus Weihrauch. *Computable Analysis: An Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.
33. D. Wingate, A. Stuhlmller, and N. D. Goodman. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Artificial Intelligence and Statistics*, AISTATS,'11, 2011.
34. Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.
35. Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 2–46, 2014.