# An application of computable distributions to the semantics of probabilistic programs

Extended abstract

Daniel Huang
Harvard University

Greg Morrisett
Cornell University

## 1. Introduction

In this extended abstract, we give semantics to a core functional probabilistic programming language (PCF with pairs) based on computable distributions. *Type-2 computable distributions* admit *Type-2 computable* sampling procedures.[1] Informally, this means that we can sample from every computable distribution using a sampling *algorithm* that operates on input bit-randomness (*e.g.*, a stream of fair coin flips). Hence, we do not need to assume black-box primitives that generate *real*-valued samples.

Computable distributions have been defined and studied in the context of Type-2 (Turing) machines and algorithmic randomness (*e.g.* [2]). Their implications for probabilistic programs have also been hinted at in the literature (*e.g.* [1]). Our contribution is to recast computable distributions and their results in the context of high-level probabilistic languages for Bayesian inference.

## 2. Background

We briefly introduce Type-2 computability (Type Two Theory of Effectivity) at an informal level (see [4] for more background). In essence, Type-2 computability studies uncountable sets by encoding elements of that set as the limit of a sequence of elements from a simpler (*e.g.*, dense) countable set. An element is then called *computable* if there is an algorithm that can enumerate the sequence, given an enumeration of the simpler set.

As an example, consider the uncountable set of reals $\mathbb{R}$ and the countable set of rationals $\mathbb{Q}$. Following the above, a real $x \in \mathbb{R}$ is computable if there is an algorithm that enumerates a subsequence of rationals $(q_{i_n})_{n\in\mathbb{N}}$ (for a fixed numbering of $\mathbb{Q}$) that converges to $x$ via the Euclidian metric. For example,

$$\pi = 3 + \sum_{k=1}^{\infty} \frac{4}{2k \cdot (2k+1) \cdot (2k+2)}$$

is a computable real number because there is an algorithm for generating arbitrarily close rational approximations to $\pi$, namely the one truncating the sequence above at some natural $N$.

A structure that captures computability on uncountable sets of interest in the probabilistic setting is the computable metric space. Indeed, a computable distribution can be seen as a point in an appropriate computable metric space with the Prokhorov metric [2]. For example, the sequence below converges to the standard Uniform distribution on the interval $(0, 1)$.

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

[1] 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.

In this example, the countable set used to approximate a point in the space of distributions on reals is the set of finite, discrete distributions that take on rational values with rational probabilities.

We can also think of computable distributions as sampling algorithms. To see this, we sketch an algorithm that samples from the standard Uniform distribution below. The idea is to generate a value that can be queried for more precision instead of a *real* value in its entirety. For instance, the following decreasing sequence of intervals $[0, 1], [0, 1/2], [1/4, 1/2], ...$ can be generated by bisecting the previous interval and using a coin flip to determine whether to take the left or right half. In the limit, we obtain a single point corresponding to the sample. 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.

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

## 3. Language

Now that we have a brief understanding of computable distributions, we give a core language and its semantics. We will use largely standard denotational semantics (see [5]).

**Syntax** The core language extends a basic functional language (PCF with pairs) with reals (constants $c$) and distributions (constants $d$).

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

The primitives $\oplus$ are (computable) operations on reals. The expressions `return` $e$ and $x \leftarrow e_1\ ;\ e_2$ correspond to return and bind in the sampling monad. The type `Real` refers to reals and the type `Samp` $\alpha$ refers to distributions. The typing rules are standard. The type `Samp` $\alpha$ is well-formed if values of type $\alpha$ support the operations required of a computable metric space (*e.g.*, naturals $\mathbb{N}$, reals $\mathbb{R}$, products of computable metric spaces, and etc.).

**Interpretation of types** The interpretation of types $\mathcal{V}[\![\cdot]\!]$ is defined by induction on types and denotes into complete partial orders (CPO's). We will introduce some notation and definitions first, ignoring computability for the moment.

We write $\text{Disc}(X)$ to equip set $X$ with the discrete order and $D_\perp$ to create a lifted domain with underlying set $\{\lfloor d \rfloor \mid d \in D\} \cup \{\perp\}$. The continuous function space between CPO's $D$ and $E$ is written $D \Rightarrow E$. $D \times_F E$ is the CPO with underlying set $\{(d, e) \in D \times E \mid F(e) = d\}$ and product ordering, where $F$ is a continuous function. Next, we introduce terminology needed to interpret `Samp` $\alpha$.

One way to think of a distribution is a *measure*, *i.e.*, a function that assigns to each (*measurable*) set a probability (*i.e.*, $[0,1]$) satisfying some properties (*e.g.*, countable-additivity). In terms of CPO constructions, every (Borel) measure $\mu$ can be restricted to an *$\omega$-continuous valuation* $\mu|_{\mathcal{O}(\mathcal{X})} \in [\mathcal{O}(\mathcal{X}) \Rightarrow [0,1]^\uparrow]$, where $\mathcal{O}(\mathcal{X})$ are the opens of the space $\mathcal{X}$ ordered by set inclusion $\subseteq$, and $[0,1]^\uparrow$ is the interval $[0,1]$ ordered by $\leq$.[2] Valuations have been used extensively in the study of probabilistic powerdomains (*e.g.* [3]). Unlike the probabilistic powerdomains, which put valuations on the open sets obtained from a CPO (via the Scott-topology), we will use the open sets induced by a computable metric space. For a computable metric space $X$, we will write $X^\perp$ to refer to the CPO $\mathrm{Disc}(X)_\perp$ (*i.e.*, the topology's specialization preorder) and $\mathcal{O}^M(\cdot)$ to reference the opens of the computable metric space.

We now give an interpretation of types. We use a standard call-by-name interpretation for the standard PCF types, and hence, omit them here. We give the interpretation of `Real` and `Samp` $\alpha$ below, where $\mathcal{V}^M[\![\cdot]\!]$ maps a type to the underlying set of a computable metric space).

$$\mathcal{V}[\![\texttt{Real}]\!] \triangleq \mathrm{Disc}(\mathbb{R})_\perp$$

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

The interpretation of `Real` is the reals with discrete order. The interpretation of `Samp` $\alpha$ is a valuation paired with a sampling function realizing it (related by the push-forward $\mathtt{psh}_\alpha$). This explicitly relates an analytic description of a distribution with a sampler.

Now, we restrict the interpretation of types to consider Type-2 computability. The semantics for this fragment can be faithfully implemented in a standard programming language.

$$\mathcal{V}^c[\![\texttt{Real}]\!] \triangleq \{r \in \mathcal{V}[\![\texttt{Real}]\!] \mid r \text{ is Type-2 computable or } r = \perp\}$$

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

The restriction for distributions essentially considers those that have Type-2 computable sampling functions, instead of any continuous sampling function.[3]

**Semantics** The semantics $\mathcal{E}[\![\Gamma \vdash e : \alpha]\!]\rho \in \mathcal{V}[\![\alpha]\!]$ is defined by induction on the typing derivation, but we will omit the judgement for brevity. It is parameterized by a global environment $\Upsilon^S$ (also omitted) that maps constants $c$ and primitive distributions $d$. The semantics is the same for the Type-2 computable version, but we need to restrict the constant reals $c$ and primitive distributions $d$ in our global environment to Type-2 computable ones. We will use the following notation 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. We use the function $\mathrm{lift}(d) = \lfloor d \rfloor$ to lift a value. Finally, the notation let $x = e_1$ in $e_2$ is a strict let binding. We go over the semantics for return and bind.

$$\mathcal{E}[\![\Gamma \vdash \texttt{return}\ e : \texttt{Samp}\ \alpha]\!]\rho \triangleq (\lambda U.\ \mathbb{1}_U(\mathcal{E}[\![e]\!]\rho), \lambda u.\ \lfloor \mathcal{E}[\![e]\!]\rho \rfloor)$$

$$\mathcal{E}[\![\Gamma \vdash x \leftarrow e_1\ ;\ e_2 : \texttt{Samp}\ \alpha_2]\!]\rho = (\lambda U.\ \int h_U\ d\mu, g^\dagger \circ f)$$

where

$$\begin{aligned} \mu &= \pi_1(\mathcal{E}[\![e_1]\!]\rho) \\ h_U &= \lambda v.\ \pi_1(\mathcal{E}[\![e_2]\!]\rho[x \mapsto v])(U) \\ f &= \lambda u.\ \text{let } v = \pi_2(\mathcal{E}[\![e_1]\!]\rho)(u_e) \text{ in } \lfloor (v, u_o) \rfloor \\ g &= \lambda(v,u).\ \pi_2(\mathcal{E}[\![e_2]\!]\rho[x \mapsto v])(w)\ . \end{aligned}$$

$u_e$ and $u_o$ refer to the even and odd bits of $u$ respectively.

The denotation of `return` $e$ is a point mass valuation centered at $e$, which corresponds to a sampler that ignores the input bit-randomness and returns $e$. The meaning of $x \leftarrow e_1\ ;\ e_2$ also gives a valuation and a sampler. In the former, we reweigh $e_2$ according to the valuation $e_1$. In the sampling view, we run the sampler denoted by $e_1$ on the input $u$ to produce a sample $v$ (a point in the computable metric space $\mathcal{V}^M[\![\alpha]\!]^\perp$) and the unused randomness $w$. Then, we run the sampler $\pi_2(\mathcal{E}[\![e_2]\!]\rho[x \mapsto v])$ with $x$ bound to $v$ on the bit-randomness $w$. We can check that the valuation and the sampling algorithm are related. For instance, for the meaning of $x \leftarrow e_1\ ;\ e_2$, we can show a composition of sampling functions corresponds to a reweighing by an integral.

The Type-2 computable sampling semantics can be provided as a Haskell library. `CMet` is a type-class that provides operations on a computable metric space, and `State` is the state monad.

```
module CompDistLib (sampler) where
type RndBits = Nat -> Bool
type Samp = State RndBits
sampler :: (CMet a) => (RndBits -> a) -> Samp a
sampler f = State (\u -> let (u1, u2) = split u
                         in return (f u1, u2))
    where
      split u = (\n -> u (2*n),\n -> u (2*n+1))
```

The type `Samp` $\alpha$ can be implemented as an instance of the State monad threading a stream of bits `RndBits`. Hence, return ignores the input bit-stream because it is a deterministic computation, and bind consumes the bit-stream to produce a sample and an unused portion of the bit-stream. The function `sampler` converts an input sampling function into the sampling monad.

**Conditioning** The language does not have a conditioning primitive built-in. Instead, we can simply write a program (in a Turing-complete language) that computes the conditional distribution—a Type-2 computable function can be programmed. Hence, the semantics of conditioning can be given as a library function.

Ackerman *et al.* show that conditioning is not computable in general, although they do identify situations in which conditioning is computable [1]. For example, conditioning is computable in the presence of independent, bounded noise. The conditioning algorithm they propose computes Bayes rule (see [1] for more details).

## 4. Conclusion

We have presented a semantics for a core functional, probabilistic programming language using largely standard techniques. We hope that the use of computable distributions captures the intuition of the semantics of a probabilistic program as a (realizable) sampler that is easily relatable to a denotational view.

## References


[1] Nathanael Leedom Ackerman, Cameron E Freer, and Daniel M Roy. Noncomputable conditional distributions. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pages 107–116. IEEE, 2011.

[2] Mathieu Hoyrup and Cristóbal Rojas. Computability of probability measures and martin-löf randomness over metric spaces. *Information and Computation*, 207(7):830–847, 2009.

[3] Claire Jones and Gordon D Plotkin. A probabilistic powerdomain of evaluations. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on*, pages 186–195. IEEE, 1989.

[4] Klaus Weihrauch. *Computable analysis: an introduction*. Springer Science & Business Media, 2000.

[5] Glynn Winskel. *The formal semantics of programming languages: an introduction*. 1993.


[2] Computable distributions restricts us to distributions on spaces with topological structure, hence Borel measure.

[3] All Type-2 computable functions are continuous [4].