Affine Monads and Lazy Structures for Bayesian Programming

SWARAJ DASH, YOUNESSE KADDAR, HUGO PAQUET, and SAM STATON, University of Oxford, UK

We show that streams and lazy data structures are a natural idiom for programming with infinite-dimensional Bayesian methods such as Poisson processes, Gaussian processes, jump processes, Dirichlet processes, and Beta processes. The crucial semantic idea, inspired by developments in synthetic probability theory, is to work with two separate monads: an affine monad of probability, which supports laziness, and a commutative, non-affine monad of measures, which does not. (Affine means that $T(1) \cong 1$.) We show that the separation is important from a decidability perspective, and that the recent model of quasi-Borel spaces supports these two monads.

To perform Bayesian inference with these examples, we introduce new inference methods that are specially adapted to laziness; they are proven correct by reference to the Metropolis-Hastings-Green method. Our theoretical development is implemented as a Haskell library, LazyPPL.

CCS Concepts: • **Theory of computation → Semantics and reasoning**; • **Mathematics of computing → Probability and statistics**.

Additional Key Words and Phrases: probabilistic programming, quasi-Borel spaces, synthetic measure theory, Bayesian inference, nonparametric statistics, categorical semantics, commutative monads, laziness, Haskell.

ACM Reference Format:
Swaraj Dash, Younesse Kaddar, Hugo Paquet, and Sam Staton. 2023. Affine Monads and Lazy Structures for Bayesian Programming. *Proc. ACM Program. Lang.* 7, POPL, Article 46 (January 2023), 31 pages. https://doi.org/10.1145/3571239

1 INTRODUCTION

Bayesian statistical models often naturally involve infinite-dimensional spaces, and in this paper we show that these can be dealt with programmatically using lazy structures. To show this, we provide a monadic metalanguage for probabilistic programming that admits streams and other lazy data structures (§2). The general key point is that an 'affine' monad can support lazy programming, but a non-affine one cannot (see §1.3 and Theorems 2.1 and 2.2). For probabilistic programming, we thus consider two monads: an affine monad of probability, and a non-affine monad of measures. We demonstrate the expressive compositional power of this through a wealth of examples (§1.2, §3, §4; [Dash et al. 2022b]). We show that these are feasible by giving new Metropolis-Hastings inference algorithms (§1.4, §6, §7) that work in a lazy setting. Our development is motivated by new compositional methods in categorical measure theory, such as quasi-Borel spaces: in Section 5 we give a new formulation of this together with an implementation as a Haskell library, LazyPPL [Dash et al. 2022b].

Authors' address: Swaraj Dash; Younesse Kaddar; Hugo Paquet; Sam Staton, University of Oxford, Oxford, UK.

2475-1421/2023/1-ART46
https://doi.org/10.1145/3571239

1.1 Monte Carlo methods, Bayesian models, and unnormalized measures

It is often said that Monte Carlo methods are the reason for the explosion in practical Bayesian statistics over the past 30 years (e.g. [Geyer 2011, §1.1], [Ghosal and van der Vaart 2017, §1.4]). One account of Monte Carlo methods is that they are methods for sampling from a probability distribution that is specified as an *unnormalized* measure, that is, a measure that is only specified up to an unknown normalizing constant (e.g. [Geyer 2011; Tierney 1994]). This matches the three primitive aspects of Bayesian statistics, which are:

- prior — a probability measure;
- likelihood — often expressed by a density, or weight, contributing to the unnormalized aspect of the measure;
- posterior — a probability measure that is proportional to the product of the likelihood and the prior, which is what the Monte Carlo method provides samples from.

Our aim here is to explore the role of laziness in building and composing these measures. Our motivation comes from two directions: practical and theoretical.

On the practical side, probabilistic programming languages for Bayesian modelling (such as Bugs [Lunn et al. 2009], Church [Goodman et al. 2008], Stan [Carpenter et al. 2017] and others) can often be regarded as programming languages describing unnormalized measures, that are endowed with efficient Monte Carlo samplers. Many focus on finite dimensional models, but some allow unbounded dimension, notably Church, which is a key starting point for our work. (See §8 for a fuller discussion of prior work.)

On the theoretical side, researchers have recently proposed categorical or synthetic accounts of probability theory [Cho and Jacobs 2019; Fritz 2020] and measure theory [Heunen et al. 2017; Kock 2012; Ścibior et al. 2018], with the aim of developing compositional structures based on commutative and affine monads and monoidal categories. There are various aims in that work, some axiomatic, and some seeking to sidestep cumbersome issues with measure theory, such as the absence of function spaces and of a strong monad of measures (see §5, where we treat this). In this way, probabilistic programming can be viewed as a *practical measure theory*, with compositionality built in, and where types are spaces, and programs are suitably good measures on the spaces. By exploring fully expressive probabilistic programming languages, we are exploring the abstract and higher-order spaces of synthetic measure theory.

1.2 Practical illustration: the Poisson process

To illustrate further on the practical side, we briefly consider a 'non-parametric' model now: the one-dimensional homogenous Poisson point process. This is a random countable collection of points on the positive real line, such that within any finite interval $[a, b]$ the expected number of points is proportional to $(b - a)$, and the number of points in disjoint regions is independent. Some draws from a Poisson point process are shown in Figure 1 (a). A Poisson point process is easy to define using laziness, and we flesh out this definition in Section 3.1.

Of course, the pictures in Figure 1 (a) each show a finite number of points, but this is because we have constrained the viewport to a finite window. In practice we may want to use the point process as part of a larger model, and in Section 3.2 we use it as part of a regression problem. Then it is unclear where to truncate it to an arbitrary viewport in advance, and, as we demonstrate, this can break the compositionality. This is often the case in statistical models, as in other areas of programming: if we just focus on running whole programs, we lose perspective of the conceptual and practical building blocks. We illustrate this further with other examples of non-parametric processes including Dirichlet process clustering (§3.3) and Gaussian process regression (§4.1).

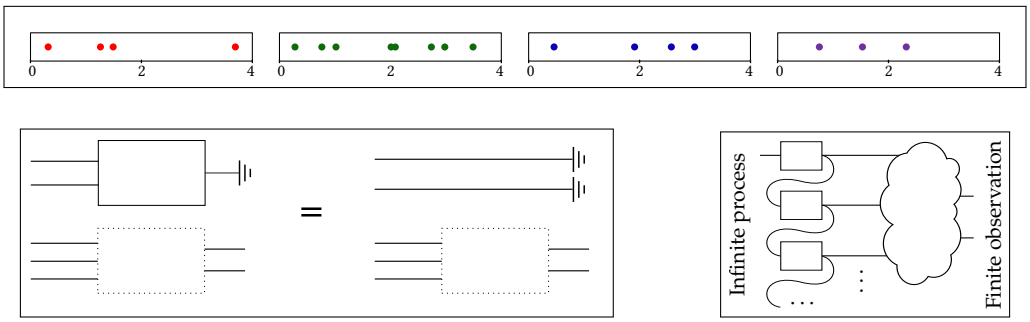


Fig. 1. (a) Four samples from a 1D Poisson point process with rate 1, with viewport restricted to $[0, 4]$. (b) The law for affine monoidal categories in string diagram form. (c) Visualizing dataflow in a lazy infinite process.

1.3 Theoretical aspects: affine monoidal structure and synthetic spaces

To illustrate the theoretical side, we recall that in the categorical foundations of measure theory, a morphism $X \to I$ into the monoidal unit describes a parameterized measure on the one-point space. If we focus on normalized probability measures, there should be exactly one measure on the one point space. So in the normalized setting, I should be a terminal object, in other words, we are working with an affine monoidal category (e.g. [Cho and Jacobs 2019; Coecke 2014; Fritz 2020; Fritz et al. 2021; Jacobs 2011; Shiebler 2020]). This is shown diagrammatically in Figure 1 (b), and we can regard the diagram as a dataflow diagram. To see where laziness comes in, we regard the Poisson point process again, now as a dataflow diagram (Fig. 1 (c)). The infinite process is on the left, and the cloud represents the plotting routine, or whatever happens next; intuitively, those morphisms on the left that are not used in what follows need never be inspected, and in that case the process can be truncated via Fig. 1 (b). Thus, affine monoidal categories are related to laziness.

As is well known, we can program with monoidal categories using monads (e.g. §5.1). The key conceptual contribution of this paper is the observation that we should program with two separate monads: an affine monad `Prob` of probability measures, allowing laziness, and a non-affine monad `Meas` of unnormalized measures, allowing the Bayesian Monte Carlo methods (§1.1). This is the basis of our metalanguage in Section 2. Theorem 2.1 demonstrates the way that lazy structures work with probability, in terms of infinite streams of samples. Proposition 2.2 shows that, once we have this functionality, we have to separate `Prob` and `Meas` to avoid deciding the halting problem.

When we regard types in probabilistic programming languages as spaces of synthetic measure theory, we see spaces from non-parametrics, such as function spaces and infinite lists, behaving intuitively and straightforwardly, even though they can be subtle from the traditional measure-theoretic approach. We give a semantic model of our metalanguage in quasi-Borel spaces (Section 5), but a novelty here is to fix the basic probability space to a space of lazy rose trees (§6.2). This lends itself to an implementation in the form of our Haskell library LazyPPL [Dash et al. 2022b].

1.4 New Metropolis-Hastings-based inference algorithms

To experiment with these examples involving laziness, we introduce new inference algorithms that build on earlier inference methods for probabilistic programming (e.g. [Wingate et al. 2011]). These take as an argument a program describing an unnormalized measure, and produce a stream of samples as output.

Traditional uses of Monte Carlo algorithms often assume a finite-dimensional state space; sometimes they can adapt to changing dimensions (e.g. [Green 1995]). But in a purely lazy setting,

Types: $\texttt{a}, \texttt{b} ::= \texttt{RealNum} \mid () \mid (\texttt{a},\texttt{b}) \mid \texttt{a} \to \texttt{b} \mid \texttt{Prob a} \mid \texttt{Meas a} \mid \ldots$

Terms: $\texttt{t}, \texttt{u} ::= \texttt{x} \mid \backslash\texttt{x} \to \texttt{t} \mid \texttt{t u} \mid \texttt{do } \{\texttt{x} \leftarrow \texttt{t ; u}\} \mid \texttt{return t} \mid \ldots$

Typing judgement ($\Gamma \vdash \texttt{t} :: \texttt{a}$)*:*

$$\frac{-}{\Gamma, \texttt{x}::\texttt{a}, \Gamma' \vdash \texttt{x} :: \texttt{a}} \qquad \frac{\Gamma, \texttt{x}::\texttt{a} \vdash \texttt{t} :: \texttt{b}}{\Gamma \vdash \backslash\texttt{x} \to \texttt{t} :: \texttt{a} \to \texttt{b}} \qquad \frac{\Gamma \vdash \texttt{t} :: \texttt{a} \to \texttt{b} \quad \Gamma \vdash \texttt{u} :: \texttt{a}}{\Gamma \vdash \texttt{t u} :: \texttt{b}}$$

$$\frac{\Gamma \vdash \texttt{t} :: \texttt{a}}{\Gamma \vdash \texttt{return t} :: \texttt{m a}} \qquad \frac{\Gamma \vdash \texttt{t} :: \texttt{m a} \quad \Gamma, \texttt{x}::\texttt{a} \vdash \texttt{u} :: \texttt{m b}}{\Gamma \vdash \texttt{do } \{\texttt{x} \leftarrow \texttt{t ; u}\} :: \texttt{m b}}\ \texttt{m} \in \{\texttt{Prob}, \texttt{Meas}\}$$

Typed constants:

$\texttt{() :: ()} \quad \texttt{(,) :: a} \to \texttt{b} \to (\texttt{a},\texttt{b}) \quad \texttt{fst :: (a,b)} \to \texttt{a} \quad \texttt{snd :: (a,b)} \to \texttt{b}$

$\texttt{0,1 :: RealNum} \quad \texttt{* :: RealNum} \to \texttt{RealNum} \to \texttt{RealNum} \quad \ldots$

$\texttt{sample :: Prob a} \to \texttt{Meas a} \quad \texttt{score :: RealNum} \to \texttt{Meas ()}$

Fig. 2. Summary of the types and terms of the monadic metalanguage.

the distributions are implicitly infinite-dimensional, and indeed our basic probability space is the infinite dimensional space of rose trees. To resolve this, we provide new instantiations of the Metropolis-Hastings-Green algorithm, that do apply in this setting, and which we have also implemented in Haskell [Dash et al. 2022b].

- Our main algorithm (§6) is purely lazy. It operates lazily over the entire infinite-dimensional state space, mutating different parts at random.
- Our other algorithms work by mixing kernels (§7). In particular, we are able to implement roughly the algorithm of [Wingate et al. 2011], adapted to this lazy setting, by using Haskell internals (`ghc-heap`) to identify which dimensions are actually being used in a given run of the program.

We can show that these algorithms are correct (via Theorems 6.2, 6.3, 7.1). Generally speaking, general purpose algorithms such as these will not work as efficiently as hand-crafted inference methods for specific scenarios. Nonetheless, they are useful for prototyping the numerous examples we consider in this paper to illustrate laziness and monads in probabilistic programming (§3, §4).

Acknowledgements. We have benefited from many helpful discussions, including with Victor Blanchi, Reuben Cohn-Gordon, Cameron Freer, Ohad Kammar, Dan Roy, Adam Ścibior, Matthijs Vákár, Frank Wood, Hongseok Yang, Mathieu Huot and other colleagues in Oxford. Thanks also to anonymous reviewers. We also benefited from presenting aspects of this work at various venues, including ACT, AIPLANS, FSCD, LAFI, PROBPROG and an early version in taught courses in OPLSS 2019 and Oxford (BSPP 2020). Research supported by AFOSR award number FA9550-21-1-0038; the ERC BLAST grant; and a Royal Society University Research Fellowship.

2 A MONADIC METALANGUAGE FOR PROBABILITY AND MEASURE

The idea of Monte Carlo based inference is that we define an unnormalized measure, by weighting different random choices, and then Monte Carlo inference provides samples from the normalized form of this measure. In this section, we encapsulate this in programming terms by using two monads, describing normalized and unnormalized measures. To make this formal, we set up an instance of Moggi's monadic metalanguage ([Moggi 1991]) outlined in Figure 2. We discuss the syntax now, with a first example in Section 2.1. We then provide a basic equational theory (§2.2) and use it to show that the separation between these monads is crucial for exploring lazy data

structures (§2.3, Theorem 2.1 and Proposition 2.2). Throughout the section, we consider simple extensions of the metalanguage that support lazy structures.

In the metalanguage, there is a distinguished type `RealNum`, thought of as the real numbers, and there are two monads:

- A probability monad `Prob` (in green font) so that `(Prob a)` intuitively contains probability measures on `a`. We typically have stock probability measures, such as `uniform :: Prob RealNum` and `normal :: RealNum → RealNum → Prob RealNum`, but we do not need to postulate these at this point.
- A measures monad `Meas` (in red font), so that `(Meas a)` intuitively contains unnormalized measures on `a`.

There are two key operations:

- `sample :: Prob a → Meas a`, which allows us to regard a probability measure as an unnormalized measure;
- `score :: RealNum → Meas ()`, which provides a measure with given weight on a single point; this is an unnormalized measure unless the weight is 1.

The `score` operation is often used with a probability density, to incorporate the likelihood in a Bayesian scenario. One can use all kinds of distributions for observations, using their densities. For example, to incorporate a Bayesian observation of data point `x` from a normal distribution with mean μ and standard deviation σ, we write `score (normalPdf μ σ x)`, where $(\texttt{normalPdf}\ \mu\ \sigma\ \texttt{x}) = e^{-(\texttt{x}-\mu)^2/(2\sigma)^2}/(\sigma\sqrt{2\pi})$. We use a Haskell notation for the metalanguage; we have implemented the metalanguage in Haskell (see §5.7, §6, [Dash et al. 2022b]) and so all the examples can be run.

2.1 First example: Bayesian linear regression

We illustrate the metalanguage with a simple 1-dimensional Bayesian regression model. (See §3 for further illustrations.) The problem of regression is that we have some data points observed, and we want to know which function generated those points. Bayesian regression does not produce one single 'line of best fit', but rather a probability distribution over the functions that might have generated the points. We start with a fairly uninformative prior distribution over linear functions, incorporate the likelihood of the observations, and produce a posterior by Monte Carlo simulation.

In statistical notation, we might define the prior by writing

$$a \sim \mathcal{N}ormal(0,3) \qquad b \sim \mathcal{N}ormal(0,3) \qquad f(x) = ax + b$$

Here, the slope `a` and intercept `b` are both drawn from normal distributions.

In the metalanguage (extended with mild syntactic sugar) we can write

```
linear :: Prob (RealNum → RealNum)
linear = do { a ← normal 0 3 ; b ← normal 0 3 ; let f x = a*x + b ; return f }
```

We do not want to assume that the data points are exactly colinear, and so we do not want observe the likelihood of the data points being exactly `f x`. Rather, we use a likelihood for the points being normally distributed around `f x`, for some small standard deviation σ. The inference problem might be written in statistical notation as:

$$f \sim \mathcal{L}inear \qquad y_i \sim \mathcal{N}ormal(f(x_i), \sigma) \qquad \text{What is } P(f|(y_i = d_i)_i)?$$

where d_i are the observed values at x_i. To this end, we define a general purpose function `regress`, which takes a standard deviation σ, a prior over the function space `prior`, and a list of `(x,y)` observations `dataset`. For convenience, we add a type of lists, and routines for operating over lists.

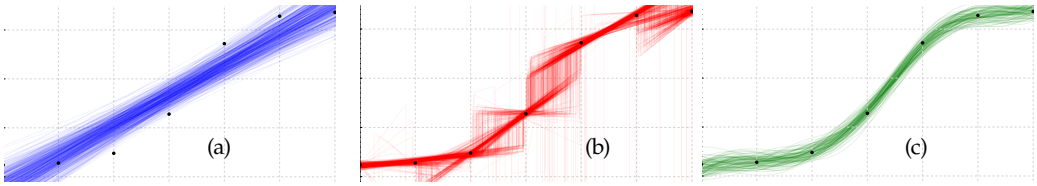


Fig. 3. Bayesian regression in LazyPPL for the data set indicated by the dots. We illustrate the posteriors starting from three different priors on the function space. From left to right: (a) linear (§2.1), (b) piecewise linear (§3.2), and (c) Gaussian processes (see §4.1).

```
regress :: RealNum → Prob (a → RealNum) → [(a,RealNum)] → Meas (a → RealNum)
regress σ prior dataset =
  do {f ← sample prior; forM_ dataset (\(x,d) → score (normalPdf (f x) σ d)); return f}
```

So linear regression in particular is achieved by performing Monte Carlo inference on the program (`regress 0.1 linear dataset`), see Figure 3 (a). We discuss inference in Section 6.

2.2 Equational reasoning

To begin to look more formally at the meaning of the programming syntax, we now consider equations between programs. We give a denotational model in Section 5. One could also give an operational semantics, and we give an implementation in Section 5.7. Either way is slightly technical, so for now we work axiomatically, by listing some equations that we intend to hold. The idea is that the equations are sound, in that any programs that are derivably equal should be equal in any good denotational interpretation, or observationally equivalent for an operational semantics.

The equational theory is not intended to be a complete system, nor is it intended as a definitional semantics by rewriting, although the equations here are useful in practice, for example as compiler transformations.

We begin with the standard equational theory of the monadic metalanguage (Figure 4), which is sound for strong monads on a cartesian closed category. Our interface has two monads, and we now state some additional equations that they should satisfy.

First, we impose that `sample :: Prob a → Meas a` should be a monad morphism, i.e.:

$$\begin{aligned} \texttt{sample (return t)} &= \texttt{return t :: Meas a} \\ \texttt{sample (do \{x ← p ; q\})} &= \texttt{do \{x ← sample p ; sample q\} :: Meas b} \end{aligned} \tag{1}$$

for `t :: a`, `p :: Prob a`, `q :: Prob b`.

The measures monad should satisfy commutativity (e.g. [Kock 1970]): for `mx :: Meas a` and `my :: Meas a`,

$$\texttt{do \{x ← mx ; y ← my ; return (x,y)\}} = \texttt{do \{y ← my ; x ← mx ; return (x,y)\}} \tag{2}$$

where $\texttt{x} \notin \text{freevars}(\texttt{my})$, $\texttt{y} \notin \text{freevars}(\texttt{mx})$. The probability monad should satisfy commutativity, and also affinity (e.g. [Jacobs 1994; Kock 1971]):

$$\text{if } \texttt{mx :: Prob a}\text{, then: } \texttt{do \{x ← mx ; return ()\}} = \texttt{return ()} \tag{3}$$

Finally, scores should combine multiplicatively, and a score of 0 should trivialize:

$$\begin{aligned} &\texttt{score 1 = return () :: Meas ()} \qquad \texttt{do \{score 0 ; mx\}} = \texttt{do \{score 0 ; my\} :: Meas a} \\ &\texttt{do \{score r ; score s\}} = \texttt{score (r * s) :: Meas ()} \end{aligned} \tag{4}$$

for `r,s :: RealNum` and `mx,my :: Meas a`.

Typed equational theory: we define a relation $\Gamma \vdash \texttt{t} = \texttt{u} :: \texttt{a}$ over typed terms $\Gamma \vdash \texttt{t}, \texttt{u} :: \texttt{a}$.

Mostly the type and environment are clear from context; the exception is the unit axiom:

If $\Gamma \vdash \texttt{t} :: ()$ then $\Gamma \vdash \texttt{t} = () :: ()$

Other axioms:
$\texttt{do}\ \{\texttt{y} \leftarrow \texttt{do}\ \{\texttt{x} \leftarrow \texttt{s}\ ;\ \texttt{t}\}\ ;\ \texttt{u}\} = \texttt{do}\ \{\texttt{x} \leftarrow \texttt{s}\ ;\ \texttt{do}\ \{\texttt{y} \leftarrow \texttt{t}\ ;\ \texttt{u}\}\}$ $[\texttt{x} \notin \mathrm{fv}(\texttt{u})]$

$\texttt{do}\ \{\texttt{x} \leftarrow \texttt{return}\ \texttt{t}\ ;\ \texttt{u}\} = \texttt{u}[\texttt{t}/\texttt{x}]$ $\quad$ $\texttt{do}\ \{\texttt{x} \leftarrow \texttt{t}\ ;\ \texttt{return}\ \texttt{x}\} = \texttt{t}$

$\texttt{fst(t,u)} = \texttt{t}$ $\quad$ $\texttt{snd(t,u)} = \texttt{u}$ $\quad$ $\texttt{t} = (\texttt{fst t}, \texttt{snd t})$ $\quad$ $(\backslash\texttt{x} \to \texttt{t})\ \texttt{u} = \texttt{t}[\texttt{u}/\texttt{x}]$ $\quad$ $\texttt{u} = (\backslash\texttt{x} \to (\texttt{u x}))$

Fig. 4. Basic equational theory of the monadic metalanguage. We give additional equations in (1)–(5).

2.3 Affine probability monads, laziness, and iid sequences

We now show the connection between the affine monad law (3) and lazy data structures. The reader might already glimpse a connection between (3) and laziness, since the law says that if the result `x` of a program expression `mx` is not used in what follows (left-hand side), then `mx` need not be considered at all (right-hand side).

To formally investigate this with lazy data structures, we suppose that the metalanguage is extended with types `Stream a` of streams, and standard accessor methods such as

```
hd :: Stream a → a    tl :: Stream a → Stream a    (:) :: a → Stream a → Stream a
```

together with syntactic sugar, and equations such as

$$\texttt{hd (x:xs)} = \texttt{x} \qquad \texttt{tl (x:xs)} = \texttt{xs} \qquad \texttt{(hd xs):(tl xs)} = \texttt{xs} \tag{5}$$

We also suppose that we have a mechanism `iid` for generating 'independent and identically distributed' infinite random sequences. This should satisfy the following recursive equation:

```
iid :: Prob RealNum → Prob (Stream RealNum)
iid p = do { x ← p ; xs ← iid p ; return (x : xs) }
```

We emphasise that the equation is for now regarded as an equational specification, not a recursive program definition with a given evaluation strategy. Although the recursive call appears to make an infinite number of random choices, we can deduce from the basic equational laws that it must be treated lazily:

Theorem 2.1. *For any finite sequence* $\texttt{i}_1, \ldots, \texttt{i}_n$ *of distinct indices, the following programs of type* `Prob (RealNum,...,RealNum)` *are equal:*

(1) $\texttt{do}\ \{\texttt{xs} \leftarrow \texttt{iid p}\ ;\ \texttt{return}\ (\texttt{xs !! i}_1, \ldots, \texttt{xs !! i}_n)\}$
(2) $\texttt{do}\ \{\texttt{x}_1 \leftarrow \texttt{p}\ ;\ \ldots\ ;\ \texttt{x}_n \leftarrow \texttt{p}\ ;\ \texttt{return}\ (\texttt{x}_1, \ldots, \texttt{x}_n)\}$

Proof notes. For instance (and omitting `do` for brevity):

$\texttt{xs} \leftarrow \texttt{iid p ; return (xs !! 2,xs !! 1)}$ [`iid` equation, 3 times]

$= \texttt{x}_0 \leftarrow \texttt{p ; x}_1 \leftarrow \texttt{p ; x}_2 \leftarrow \texttt{p ; xs} \leftarrow \texttt{iid p ; return ((x}_0\texttt{:x}_1\texttt{:x}_2\texttt{:xs)!!2,(x}_0\texttt{:x}_1\texttt{:x}_2\texttt{:xs)!!1)}$

$= \texttt{x}_0 \leftarrow \texttt{p ; x}_1 \leftarrow \texttt{p ; x}_2 \leftarrow \texttt{p ; xs} \leftarrow \texttt{iid p ; return (x}_2\texttt{,x}_1\texttt{)}$ [(5)]

$= \texttt{x}_1 \leftarrow \texttt{p ; x}_2 \leftarrow \texttt{p ; return (x}_2\texttt{,x}_1\texttt{)}$ [(3)]

$= \texttt{x}_1 \leftarrow \texttt{p ; x}_2 \leftarrow \texttt{p ; return (x}_1\texttt{,x}_2\texttt{)}$ [(2) and rename variables] □

To emphasise the connection with lazy structures, we make the following remarks. Our equational theory has thus far not discussed 'termination', however, in any reasonable semantics, program (2) in Theorem 2.1 would be regarded as terminating. Thus program (1), being equal to (2), should also

be regarded as terminating. Thus the call to `iid` induces an unbounded stream of results, despite terminating, which is the essence of laziness.

Using similar arguments, we can assume a more general primitive that generates a stream of results by iterating a function `b → Prob (a, b)`. This is subject to the following equation:

```
unfold :: (b → Prob (a, b)) → b → Prob (Stream a)
unfold f y = do { (x, y') ← f y; xs ← unfold f y'; return (x : xs)}
```

We note that we could define `iid` by `iid p = unfold (\_ → do {x ← p; return (x, ())}) ()`.

Discussion of problems with laziness in general measures monads. We do not ask that the general measures monad, `Meas a`, is affine, because then the `score` construct would disappear. In practice, this means that we cannot expect to construct lazy data structures using recursive definitions in the `Meas` monad, as we now explain.

Suppose for a moment that we did have the equivalent of `unfold` and `iid` in the `Meas` monad:

```
unfoldM :: (b → Meas (a, b)) → b → Meas (Stream a)
iidM :: Meas RealNum → Meas (Stream RealNum)
```
(6)

For instance, `iidM m = do { x ← m ; xs ← iidM m ; return (x : xs)}`. Consider the program `do { xs ← iidM (do {score 2 ; return 42}) ; return (hd xs)} :: Meas RealNum`. We are unable to use the equations of the monadic metalanguage to simplify this program, because we cannot use (3), and the semantics is unclear. (Possible semantics include: the program does not terminate, or, the program terminates with infinite score, since $2^n \to \infty$ as $n \to \infty$.)

It is still possible to use lazy data structures when building unnormalized measures: for instance we can write `sample (iid p) :: Meas (Stream a)` whenever `p :: Prob a`. But we show now that implementing a general `unfoldM` and `iidM` such that `sample (iid p) = iidM (sample p)` is impossible, whenever our equations hold.

Proposition 2.2. *Suppose that there are terms* `unfoldM` *and* `iidM` *as in* (6) *such that we have* `iidM (return t) = sample (iid (return t))`. *Then among the programs*

$$\{\texttt{t} \mid \exists \texttt{r}, \texttt{u} \textit{ such that } \texttt{t} = \texttt{do } \{\texttt{score r ; return u}\} :: \texttt{Meas RealNum}\}$$

the score `r` *is undecidable.*

Proof. Pick an arbitrary Turing machine M. We exhibit a program `h` such that we have either `h = score 0 ; return 0` if M halts, and `h = score 1 ; return 42` otherwise. Thus it is in general undecidable whether the score is 0 or 1. To this end, write `steps n` if M halts after simulating it for `n` steps. Suppose we extend the metalanguage with this `steps` construction.

```
h :: Meas Integer
h = hd <$> (unfoldM (\n → do{ score (if (steps n) then 0 else 1); return (42,n+1) })) 0
```

(where `(t <$> u) = do {x ← u ; return (t x)}`). Suppose that M terminates at time n. Then, unrolling `<$>` and the recursive equation for `unfoldM`, and simplifying using (1)–(5) and Fig. 4:

```
h = do {xs ← (unfoldM...) 0 ; return (hd xs)} = do {xs ← (unfoldM...) 1 ; return 42}
  =   ...   = do {xs ← (unfoldM...) n ; return 42}
  = do {score 0 ; xs ← (unfoldM...) (n+1) ; return 42} = do {score 0 ; return 0}.
```

On the other hand, if the machine M does not halt, then `h = hd <$> iidM (return 42)`. By the assumption in the theorem, `h = hd <$> sample (iid (return 42))`, and so by (1) and (4), we have

```
h = hd <$> do {xs ← sample (iid (return 42)) ; return (42 : xs) }        [iid & (1)]
  = do {xs ← sample (iid (return 42)) ; return 42}                       [<$> & (4)]
  = sample (do {xs ← iid (return 42) ; return 42}) = sample (return 42)  [(1) & (3)]
  = return 42 = do {score 1 ; return 42}                                 [(1) & (4)]
```

□

If the score is undecidable, this means the system is not amenable to Monte Carlo simulation, because the score is needed to know whether or not to reject a run of the program. For the example in the proof of Proposition 2.2, a Monte Carlo simulation would have to always reject if the machine M terminates, and always accept if the machine M does not terminate, so it would have to decide the Halting problem, which is absurd. There are other decidability problems in probabilistic programming (e.g. [Ackerman et al. 2019; Huang et al. 2018]), but this barrier occurs even without any statisticial primitives.

In summary, we can program random lazy data structures using the affine `Prob` monad and constructions like `iid`, and execution must treat them lazily (Theorem 2.1). By contrast, we cannot hope to program lazy data structures using the `Meas` directly, for decidability reasons (Proposition 2.2). Nonetheless we can build random lazy data structures using the `Prob` monad, and constructions such as `iid`, and then regard them as arbitrary measures using `sample :: Prob a → Meas a`.

2.4 Comparison with prior work on synthetic measure theory

Previous work considered a synthetic measure theory based on a cartesian closed category with countable sums and products and a commutative monad M that is countably additive [Kock 2012; Ścibior et al. 2018]. We can then construct an affine submonad $P \subseteq M$ as an equalizer, and an object of scalars $R = M(1)$. Any such model is a model of the metalanguage in Sections 2.2: `Prob` $= P$, `Meas` $= M$, `RealNum` $= R$, `sample` is the inclusion, and `score` is the identity function.

This connects to other axiomatizations too: for a model of synthetic measure theory, the Kleisli category of M is a CD/GS-category [Cho and Jacobs 2019; Fritz and Liang 2022; Stein 2021b], and the Kleisli category of P is a Markov category [Fritz 2020; Fritz et al. 2020]. Infinite processes such as `iid` have recently been studied in Markov categories [Fritz and Rischel 2020], but not in the unnormalized setting.

One difference between our metalanguage and synthetic measure theory is that `Meas`, `Prob` and `RealNum` are given as explicit ingredients here; `Prob` is not necessarily an equalizer and `RealNum` is not necessarily equal to `Meas ()`. We found it helpful to make these distinctions in practice. For instance, the expression `do {x ← sample uniform ; score (f x)} :: Meas ()` intuitively corresponds to the `RealNum` that is the integral $\int_0^1 \mathtt{f}(x)\,dx$, but for general `f`, this can only be calculated approximately, e.g. via Monte Carlo simulation.

3 EXAMPLES TAKING ADVANTAGE OF LAZINESS

We now give examples of probabilistic programs with lazy data structures, using the metalanguage from Section 2. We are able to run the programs by using our implementation (LazyPPL, §5.7, [Dash et al. 2022b]).

The examples show that laziness, and two separate monads `Prob` and `Meas`, allow compositional programming; without this, we cannot use `iid`, and we have to pass around truncation bounds.

3.1 Point processes as lazy streams of points

A random collection of points is called a *point process*. A first example of an infinite point process is a *Poisson* point process on the positive reals (see also Figure 1 (a)). In statistical notation, we consider a sequence of real random variables $x_0, x_1, \ldots$ such that the gaps are exponentially distributed:

$$x_0 \sim \mathcal{E}xponential(r) \qquad (x_{i+1} - x_i) \sim \mathcal{E}xponential(r)$$

This can be defined in a minor extension of our metalanguage, by giving a random stream [1]:

```
poissonPP :: RealNum → Prob [RealNum]
poissonPP rate = do { steps ← iid (exponential rate) ; return (scanl (+) 0 steps) }
```

This returns an infinite random stream, illustrated in Figure 1 (a). Although this is simple, it becomes powerful when we compose with other models, as we show in the following sections. (See also [Dash and Staton 2020], for more on programming with point processes.)

Comments on avoiding lazy data structures. Without using infinite lazy data types, we can only implement a truncated Poisson point process, where an upper bound is specified:

```
poissonPPbounded :: RealNum → RealNum → RealNum → Prob [RealNum]
poissonPPbounded rate lower upper = do
  step ← exponential rate
  let x = lower + step
  if x > upper then return []
  else do {xs ← poissonPPbounded rate x upper; return (x:xs)}
```

This truncated form is an obstacle to compositional modelling, because when the Poisson point process is used as part of a more complex model, we will have to calculate an upper bound and pass it around manually, as we will discuss further in Section 3.2.

(As an aside, we mention that although the bounded program will always return a finite list in practice, this relies on a mathematical argument: any sequence of draws from an exponential distribution will almost surely go above a given upper bound in a finite number of steps. This is no problem in a general purpose programming language, but may prove subtle in, say, a dependently typed language with termination guarantees.)

Of course, some truncation must happen at some point. In practice, in LazyPPL, it happens at the top level, the plotting routine stops looking at the stream beyond the viewport, and the laziness propagates from here.

3.2 Piecewise regression with lazy change points

We combine the Poisson Point Process (§3.1) with the linear regression model (§2.1), to obtain *piecewise* linear regression, where the prior is over piecewise linear functions. We define a function that will splice together different draws from a random function given a random sequence of change points:

```
spliceProb :: Prob [RealNum] → Prob (RealNum → RealNum) → Prob (RealNum → RealNum)
spliceProb pp f = do {xs ← pp ; fs ← iid f ; return $ splice xs fs }
```

Here we assume a basic deterministic function

```
splice :: [RealNum] → [RealNum → RealNum] → (RealNum → RealNum)
```

[1] Henceforth we use `[a]` instead of `(Stream a)`, recalling that Haskell lists include infinite lists.

which splices together a sequence of functions at the given change points. The sequence of change points can be infinite, so the function can have an infinite number of pieces. But this is no problem if our probability monad is affine, and this can be handled lazily.

We can perform piecewise linear regression using a Poisson point process, for example via

```
piecewiseLinear :: Prob (RealNum → RealNum)
piecewiseLinear = spliceProb (poissonPP 0.2) linear
```

and then `regress 0.1 piecewiseLinear dataset`, which gives Fig. 3 (b), using the inference in §6.

Comments on avoiding lazy data structures. Our piecewise linear model demonstrates why it is important to have separate `Prob` and `Meas` monads. As we have explained in Section 2.3, it is impossible to have a general `iid` on arbitrary measures, but it is fine on probability measures. We have used `iid` twice in this model. It is part of `spliceProb`, and so it is crucial to know that the prior `f` is a probability measure on functions.

This piecewise linear regression can be implemented without infinite lazy structures, by using `poissonPPbounded` and rewriting `spliceProb` to just sample a bounded number of functions `fs`, depending on the length of `xs`. For this first example, the bound can come from the chosen viewport, or data range, which might not be hard to calculate, but in general, propagating this is not compositional. For example, we can quickly rescale a random function on the x-axis:

```
rescale :: Prob (RealNum → RealNum) → Prob (RealNum → RealNum)
rescale p = do { f ← p ; return (\x → f(2 * x)) }
```

We can then perform regression with `regress 0.1 (rescale piecewiseLinear) dataset`. To do this without lazy structures, with explicit bounds, we would also need to rescale the bounds by hand. So compositionality with explicit bounds is impossible.

As an aside, we point out the beauty of higher-order probabilistic programming for compositional model building: it is very easy to switch the Poisson process for a different point process, or to use piecewise *constant* regression, and so on (see the repository [Dash et al. 2022b] for more examples).

3.3 Clustering using a lazily broken stick

For a set of data points, clustering is the problem of finding the most appropriate partition into clusters. In non-parametric clustering, the number of clusters is unknown and unbounded. As we demonstrate, this is closely connected to laziness. The separation into two monads, `Prob` and `Meas`, is crucial for a compositional specification in our metalanguage.

Stick-breaking. In *Bayesian* clustering, one considers a prior distribution over possible partitions of the data into clusters. This is usually described in terms of stick-breaking, where the unit interval $[0, 1]$ is broken into an infinite number of sticks, each representing a cluster, and the size of the stick is the proportion of points in that cluster.

As an example, we consider stick-breaking for a Dirichlet process (e.g. [Ghosal and van der Vaart 2017]), where at each step we break off a portion of the remaining interval according to a beta distribution. This is often written in statistical notation as

$$r_k \sim \mathcal{B}eta(1, \alpha) \qquad v_k = r_k \cdot \textstyle\prod_{i=1}^{k-1}(1 - r_i)$$

The random sequence $(v_0, v_1, \dots)$ has the property that $\sum_{i=0}^{\infty} v_i = 1$, almost surely.

Stick-breaking is easy to define as a random stream `stickBreaking` α in our metalanguage.

```
stickBreaking :: RealNum → Prob [RealNum]
stickBreaking α = do rs ← iid $ beta 1 α
                     let vs = zipWith (*) rs $ scanl (*) 1 $ map (1-) rs
```

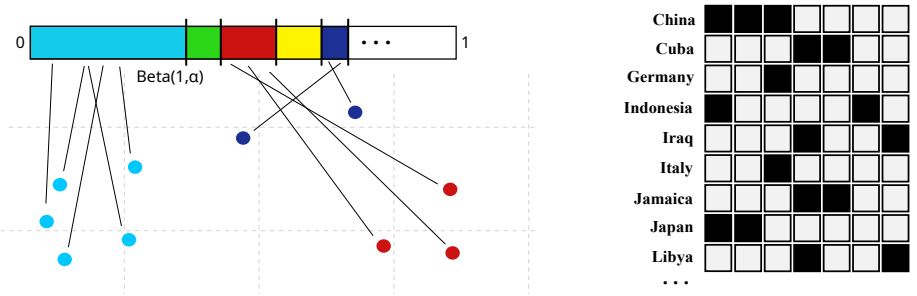


Fig. 5. From left to right: (a) Dirichlet process clustering by stick-breaking (§3.3); (b) Feature extraction for a psychological experiment (§3.4): the features (columns) are automatically inferred and often interpretable.

```
                    return vs
```

We can then regard this sequence of probabilities, which sums to 1, as a distribution on natural numbers. One way to do this is to use a uniform sample in $[0, 1]$ to index into the sequence, via elementary functional programming.

```
toProb :: [RealNum] → Prob Int
toProb vs = do {r ← uniform; return $ findIndex (> r) (scanl (+) 0 vs)}
```

For clustering, we also assign some parameters to each cluster, i.e. to each part of the stick. For example, if the clusters are located geometrically, we might associate a random position value to each cluster, assuming that the data points in this cluster follow a normal distribution around that position. The parameters are taken from a 'base' measure. The general procedure is the Dirichlet Process:

```
dp :: RealNum → Prob a → Prob (Prob a)
dp α base = do {vs ← stickBreaking α;
  xs ← iid base; return $ do {n ← toProb vs; return (xs!!n)}}
```

We can then piece together a Dirichlet Process clustering model.

```
cluster :: RealNum → (Prob a) → ((b, a) → RealNum) → [b] → Meas [(b,a)]
cluster α base likelihood dataset = do
  p ← sample $ dp α base
  xs ← sample $ iid p
  let taggedData = zip dataset xs
  mapM_ (score . likelihood) taggedData
  return taggedData
```

This model (`cluster α base likelihood dataset`) partitions the data into random clusters via stick-breaking, draws a value from `base` for each cluster, and incorporates a score for each data point using `likelihood`.

Comments on avoiding lazy data structures. Our clustering model demonstrates why it is important to have a separate `Prob` monad. We have used `iid` three times in this model. As previously mentioned, it is impossible to have a general `iid` on arbitrary measures, but it is fine on probability measures (§2.3). For example, `iid` is used in `dp`, and so it is crucial to know that the argument `base` is a *probability* measure.

In the literature, there are various ways of avoiding the infinite lazy data structure of stick breaking. One common approach is to truncate the distribution. For example, we might pick some

small ϵ, and then find n such that $(1 - \sum_{i=1}^{n} v_i) < \epsilon$, so that the chance of needing v_i with $i \geq n$ is small enough to be ignored. Care needs to be taken, then, to ensure that this ϵ is chosen suitably for the model, which is potentially difficult and non-compositional when stick-breaking is part of a complex model.

3.4 Non-parametric feature assignment

Clustering assigns a single cluster to each data point. We briefly also mention the task of *feature assignment*, which assigns a finite set of features to each data point. A feature assignment for a data set consists of a finite set of features, together with a subset of these features for each data point. For example we can have a set of movies, and a feature assignment could be a set of genres for each movie. In the non-parametric setting, the number of features is unbounded even for a fixed data set.

As an application, we consider a basic problem from applied psychology [Navarro and Griffiths 2006]. The problem is as follows. We have a set of countries together with a similarity coefficient for each pair of distinct countries, calculated based on answers from participants in a study. The goal is to infer a set of underlying features which characterize the countries in the participants' minds, and influence their judgement. The outcome is shown in Figure 5 (b), as a Boolean matrix indicating assignments of features (columns) to countries (rows). Notice that the inferred features do match actual perceived features, such as continent (column 1), industry (col. 2), development (col. 4), conflict (col. 7).

We now describe the model, and where the laziness comes in. The overall process is described as a function

```
bp :: RealNum → Prob a → Prob (Prob [a])
```

that corresponds to a combination of Beta and Bernoulli processes (e.g. [Thibaux and Jordan 2007]). This generalizes the Dirichlet process `dp :: RealNum → Prob a → Prob (Prob a)` (§3.3) by producing a list rather than a single value. See [Dash et al. 2022b] for details.

A note about implementation. The Beta process can be constructed using a variant of stick-breaking: each stick now corresponds to a feature and its size is the proportion of points with that feature [Paisley et al. 2012]. But in practice this construction can only be used as an approximation, because we can only look at a finite number of sticks and the features are unbounded. By contrast, for the Dirichlet process we can stop stick-breaking once we have found a cluster. These difficulties appear to coincide with continuity issues in the analysis of the Beta process (e.g. [Roy et al. 2013, Sl. 6], also [Ackerman et al. 2019; Roy et al. 2008; Roy 2014]).

In our implementation [Dash et al. 2022b] we avoid these issues by making use of some amount of hidden state. We set up an Indian Buffet process [Griffiths and Ghahramani 2011], which is a way to assign features to data points sequentially by keeping track of features assigned to previous points. This terminates, and gives the same model [Thibaux and Jordan 2007]. The state is safely encapsulated (see e.g. [Ackerman et al. 2016]), and we use streams to deal with the unbounded aspects.

4 FURTHER EXAMPLES WITH RANDOM FUNCTIONS

Our examples so far have focused on random streams (§3) and the `iid` construction (§2.3). The type of streams is often isomorphic to the function type `Nat → a`, regarding a stream as the function that returns its value at each index. In this section we briefly discuss the role of the affinity of the monad `Prob` in exploring function types.

We have seen random functions `linear` and `piecewiseLinear :: Prob (RealNum → RealNum)` in Section 3. We now consider Gaussian Processes (§4.1) and stochastic memoization (§4.2), as other

ways of building random functions. There are plenty more besides; in the repository we define random functions based on random programs, so that regression becomes program induction [Dash et al. 2022b].

4.1 Gaussian process regression

A Gaussian process (e.g. [Ghosal and van der Vaart 2017, Ch. 11]) is a random function f with the property that for a finite input sequence $(x_1 \dots x_n)$ the output distribution $(f(x_1) \dots f(x_n))$ is a multivariate normal distribution. For simplicity, we focus on 1-dimensional Gaussian processes, which are random functions $\mathbb{R} \to \mathbb{R}$. As with any function, we can call these with an unbounded number of arguments: n is not fixed.

A Gaussian process is parameterized by a mean function $m : \mathbb{R} \to \mathbb{R}$ and a covariance function $k : \mathbb{R}^2 \to \mathbb{R}$. Gaussian processes admit conditioning: in statistical notation, if $f \sim \mathcal{GP}(m, k)$, then

$$\text{for all } x_0 \in \mathbb{R}, \quad f(x_0) \sim \mathcal{N}ormal(m(x_0), \sqrt{k(x_0, x_0)}) \quad \text{and} \quad f|(x_0 \mapsto y) \sim \mathcal{GP}(m|_{x_0 \mapsto y}, k|_{x_0 \mapsto y})$$

where $m|_{x_0 \mapsto y}, k|_{x_0 \mapsto y}$ are respectively the conditional mean and covariance functions[2]. To formalize this in our metalanguage, we extend it with a Gaussian process primitive

```
gp :: (RealNum → RealNum) → (RealNum → RealNum → RealNum) → Prob (RealNum → RealNum)
```

and consider the following equation at type `Prob (RealNum → RealNum)`, for any x_0:

```
gp m k  =  do y ← normal (m x0) (sqrt (k x0 x0))
              f ← gp (m' x0 y) (k' x0 y)                    (7)
              return (\x → if x==x0 then y else f x)
```

where `m' x0 y` and `k' x0 y` are the conditional mean and covariance functions. To lazily evaluate a Gaussian process we can use (7) to unroll it as necessary, using different values for x_0, and then use the affine property of the probability monad (3) to remove any reference to `gp` (c.f. Theorem 2.1).

Once `gp` is in our metalanguage, we can combine Gaussian processes with other random functions. For example, we can use a random linear function (§2.1) as a mean for a Gaussian process:

```
gpWithLinear:: Prob (RealNum → RealNum)
gpWithLinear = do { f ← linear ; return $ gp f (rbf 1 1) }
```

(using the standard radial basis kernel `rbf`). The result of running `regress 0.3 gpWithLinear dataset` is a posterior over smooth curves fitting the points, shown in Figure 3 (c).

Note about implementation. Although the defining equation (7) is straightforward and reminiscent of Theorem 2.1, it is different from the other distributions so far in that it is not a Haskell-style definition. We implemented `gp` in LazyPPL [Dash et al. 2022b] by first making an infinite sequence of standard normals, using `iid (normal 0 1)`, and then by using a hidden memo table and standard linear algebra for conditional probability in Gaussian processes. In general, hidden state would violate the affine and commutativity properties; in this setting, since we validate (7), the affine and commutative properties of `Prob` (2,3) remain satisfied.

Comments on the role of lazy structures. Our Gaussian process regression example terminates because (7) allows us to regard the value of the function `f` at certain inputs, and disregard its value at unused arguments. The input values are the points of the data observations, and the points used for plotting, which are dependent on the viewport and the resolution.

Since `gp` uses the affine `Prob` monad, and not the `Meas` monad, it can be passed to the second-order distributions above from Section 3. We can write `spliceProb (poissonPP r) (gp m k)` to make a

[2] $m|_{x_0 \mapsto y}(x) = m(x) + \frac{k(x_0,x)(y - m(x_0))}{k(x_0,x_0)}$, and $k|_{x_0 \mapsto y}(u, v) = k(u, v) - \frac{k(u,x_0)k(x_0,v)}{k(x_0,x_0)}$.

piecewise Gaussian process, which amounts to what is called a 'jump process' in statistics. We can write `dp α (gp m k)` for a Dirichlet process mixture of Gaussian processes.

The function space of the metalanguage is here being used to hide a lazy process. One *could* replace the call to `gp` with a multivariate normal distribution of fixed dimension, but then the plotting points and observation points would need to be passed manually as an input to the function `gpWithLinear`. The dimension would need to be picked based on the number of data points and the plotting resolution.

With the lazy behaviour of `gpWithLinear`, we can easily rescale the function, for example using `rescale gpWithLinear :: Prob(RealNum → RealNum)`. (We can consider more exotic rescalings too, such as 'deep Gaussian processes' [Damianou and Lawrence 2013], by composing Gaussian processes.) With a non-lazy version based on explicit multivariate normal distributions, one would also need to take care to manually rescale the viewport, resolution, and the observation points, which would not be compositional.

4.2 Stochastic memoization

In standard programming, memoization is a form of laziness where a function caches previous results instead of re-calculating [Michie 1968]. In functional probabilistic programming, memoization becomes a powerful method for building infinite-dimensional probability measures (e.g. [Goodman et al. 2008; Roy et al. 2008; Wood et al. 2009]). We now discuss memoization in the setting of the metalanguage, with the separation of `Prob` and `Meas`.

In statistical terms, suppose we have a parameterized distribution, $\mathcal{P}(x)$ over a space B, with parameters x in a space A. Stochastic memoization provides a random function $F : A \to B$ with

$$\forall x \in A. \quad F(x) \sim \mathcal{P}(x).$$

We can study stochastic memoization in our metalanguage by adding a constant

```
memoize :: (a → Prob b) → Prob (a → b)
```

The idea is that `memoize p :: Prob (a → b)` randomly picks an assignment of results for each argument, informally by sampling once from `(p x)` for every `(x :: a)`. We consider the following equation at type `Prob (a → b)`:

```
memoize p = do { y ← p x₀; f ← memoize p; return (\x → if x==x₀ then y else f x)}   (8)
```

As with `gp`, if the memoized function is only called with a finite collection of arguments, we can unroll equation (8) for each of the arguments, and then use the affine law (3) to remove any reference to `memoize`.

4.2.1 Memoization on countable types. By regarding `Nat → a` as isomorphic to `Stream a`, we can define memoize with domain type `Nat` just using basic primitives for lazy monadic streams, according to the following equation:

```
memoize :: (Nat → Prob b) → Prob (Nat → b)
memoize f = do { ys ← mapM f [0..] ; return $ \x → ys !! x }
```

(In practice [Dash et al. 2022b], we define `memoize` more efficiently using tries; c.f. [Hinze 2000].) We can see, in particular, that `memoize (\_ → p)` corresponds to `iid p`. With this in mind, following Proposition 2.2, we cannot hope to have `memoizeMeas :: (a → Meas b) → Meas (a → b)`, and the restriction to the probability type is crucial.

4.2.2 Illustrations of using memoization. In Section 3.3 we have seen the Dirichlet process applied to clustering. One common method in statistics is to regard the Dirichlet process as a random measure, by applying it to a generic base measure, for example,

```
dp α uniform :: Prob (Prob RealNum)
```

That is to say, rather than assigning informative parameters to the clusters (such as mean position) from the outset, we begin by assigning each cluster a name, picked uniformly from $[0,1]$.

Later on, of course, we would like to assign a position in (say) 2D space to each cluster. Since the clusters are named by real numbers, this assignment of positions to names of clusters is a random function $f : \mathbb{R} \to \mathbb{R}^2$ such that, say,

$$\forall r.\, f_1(r) \sim \mathcal{N}ormal(0,3) \quad \& \quad f_2(r) \sim \mathcal{N}ormal(0,3).$$

This can be implemented in our metalanguage by using memoization:

```
p ← dp α uniform; xpos ← memoize (\_ → normal 0 3); ypos ← memoize (\_ → normal 0 3)
```

If we return `do {r ← p ; return (xpos r,ypos r)}`, then this whole program is equivalent to using a different base distribution, `dp α (mvnormal (0,0) (3,3))`. However, the memoized form is more compositional: we can later assign other attributes to clusters in another part of the model (see [Dash et al. 2022b]).

5 INTERPRETATION OF THE PROBABILITY MONAD USING INFINITE TREES

We now provide a model of the metalanguage from Section 2, with two monads: `Prob` and `Meas`. Our model is based on quasi-Borel spaces [Heunen et al. 2017]. The novelty here is that, in prior work on quasi-Borel spaces, one works up to lots of measurable isomorphisms, e.g. $\mathbb{R} \cong \mathbb{R}^{\mathbb{N}}$. Here we are explicit about this, and so the connection to laziness is clarified. Because we are working more concretely, a new implementation is suggested (§5.7, [Dash et al. 2022b]).

5.1 Affine and commutative monads from monoidal categories

We begin by recalling a convenient presentation of commutative monads based on monoidal categories. Recall that a symmetric monoidal category is a category C with a functor $\otimes : C \times C \to C$ and an object I, together with coherent associativity and symmetry data. It is called *affine* if I is a terminal object. The following is well known in some circles (e.g. [Levy et al. 2003]), and allows us to define commutative monads by directly defining their Kleisli categories.

Lemma 5.1. *Let $J : \mathcal{V} \to C$ be a functor that is identity on objects (i.e. $ob(\mathcal{V}) = ob(C)$).*

(1) *If J has a right adjoint R then C is the Kleisli category of the monad RJ.*
(2) *Suppose moreover that $\mathcal{V}$ has products. Then to give RJ the structure of a commutative monad (2) is to give C the structure of a symmetric monoidal category such that J is a strong symmetric monoidal functor.*
(3) *Moreover, RJ is an affine monad (3) if and only if C is affine monoidal.*

Suppose that C is the Kleisli category for a monad, say `Meas`. The composition in C corresponds to the sequencing of the metalanguage. First, a context $\Gamma = (\mathtt{x}_1 :: \mathtt{a}_1, \ldots, \mathtt{x}_n :: \mathtt{a}_n)$ is interpreted as the monoidal product $\mathtt{a}_1 \otimes \cdots \otimes \mathtt{a}_n$ in C. Then a term $\Gamma \vdash$ `t :: Meas a` can be interpreted as a Kleisli morphism, i.e. a morphism $\Gamma \to$ `a` in C. If we also have `x::a`, $\Delta \vdash$ `u :: Meas b`, then the sequencing $\Gamma, \Delta \vdash$ `do {x ← t ; u} :: Meas b` is the following composite morphism:

$$\Gamma, \Delta \xrightarrow{\mathtt{t} \otimes \Delta} \mathtt{a}, \Delta \xrightarrow{\mathtt{u}} \mathtt{b}$$

With this in mind, the commutativity law (2) amounts to the interchange law of monoidal categories, $(\mathtt{a}' \otimes g) \circ (f \otimes \mathtt{b}) = (f \otimes \mathtt{b}') \circ (\mathtt{a} \otimes g)$ (for $f : \mathtt{a} \to \mathtt{a}'$ and $g : \mathtt{b} \to \mathtt{b}'$).

We now define two monads by instantiating this lemma. In both cases, $\mathcal{V}$ is the category of quasi-Borel spaces (§5.3); C will be a category of probability (§5.4) or measure kernels (§5.5).

5.2 Key intuition: randomized functions and splitting

Let Ω be a set of random seeds. A *randomized function* between sets X and Y is a function $f\colon X \times \Omega \to Y$, that depends on the random seed. Suppose that we have a method for splitting random seeds, $\gamma : \Omega \to \Omega \times \Omega$ (e.g. [Steele Jr et al. 2014]). Then we can compose randomized functions

$$f\colon X \times \Omega \to Y \qquad g\colon Y \times \Omega \to Z \qquad (g \circ f)\colon X \times \Omega \to Z$$

by $(g \circ f)(x, \omega) = g(f(x, \omega_1), \omega_2)$, where $\gamma(\omega) = (\omega_1, \omega_2)$. This is the essence of our treatment of probability. However, put plainly like this, composition is not associative. To achieve associativity, we equate certain randomized functions, but to do this we need to talk about expected values, measures and integration (§5.3–5.5).

We aim to use Lemma 5.1 to convert this category into a Kleisli category. Indeed, by currying, we can regard a function $f\colon X \times \Omega \to Y$ as a function $X \to Y^\Omega$. Once we equate certain functions, Y^Ω becomes a monad, and we are thus in the setting of programming with monads. (We emphasise that, despite appearances, this is not the standard reader monad, and (Ω, γ) is not a comonoid.)

5.3 Rudiments of quasi-Borel spaces

5.3.1 Rudiments of measure theory. We first recall some rudiments of measure theory.

Definition 5.2. A measurable space (X, Σ_X) is a set X together with a set Σ_X of 'measurable subsets' of X, which must be a σ-algebra, i.e. closed under countable unions and complements. A measure on a space (X, Σ_X) is a function $\mu\colon \Sigma_X \to [0, \infty]$ that is σ-additive ($\mu(\biguplus_{i=1}^\infty U_i) = \sum_{i=1}^\infty \mu(U_i)$); it is a probability measure if it is normalized, i.e. if $\mu(X) = 1$. A function $f\colon (X, \Sigma_X) \to (Y, \Sigma_Y)$ is *measurable* if $f^{-1}(U) \in \Sigma_X$ for all $U \in \Sigma_Y$.

A key measurable space is $(\mathbb{R}, \Sigma_\mathbb{R})$, where $\Sigma_\mathbb{R}$ comprises the Borel sets, the least σ-algebra containing the open intervals. The unit interval $([0,1], \Sigma_{[0,1]})$ is a subspace, and the uniform measure on $[0,1]$ is a measure that assigns to each open interval its length. For any measure μ on (X, Σ_X), we can find the expected value of any measurable function $f\colon (X, \Sigma) \to (\mathbb{R}, \Sigma_\mathbb{R})$, notated $\int f(x)\, \mu(\mathrm{d}x) \in [0, \infty]$, the Lebesgue integral of f with respect to μ. Two measures are the same if they induce the same integration operator.

5.3.2 Borel spaces and quasi-Borel spaces. We begin with the notion of *standard Borel space*. In fact, we do not need the traditional definition; the following characterization will suffice.

Proposition 5.3 (e.g. [Kechris 1987]). (1) *A standard Borel space is a measurable space (X, Σ_X) that is either (a) countable, with Σ_X the powerset of X, or (b) measurably isomorphic to $(\mathbb{R}, \Sigma_\mathbb{R})$.*
(2) *Any measurable subspace of $\mathbb{R}$ is standard Borel (e.g. $[0,1]$ is standard Borel).*
(3) *Standard Borel spaces are closed under countable products.*

Let Ω be a fixed uncountable standard Borel space (traditionally $\Omega = \mathbb{R}$, but see Section 5.4.1).

Definition 5.4 ([Heunen et al. 2017]). A quasi-Borel space (X, M_X) comprises a set X together with a collection M_X of functions $\Omega \to X$, called 'admissible random elements', such that

- all constant functions are in M_X;
- composition: if $\alpha \in M_X$ and $f\colon \Omega \to \Omega$ is measurable then $(\alpha \circ f) \in M_X$;
- gluing: if $\alpha_1 \ldots \alpha_n \ldots \in M_X$ and $\Omega = \biguplus_{n=1}^\infty U_i$ measurable then $\alpha \in M_X$ where $\alpha(\omega) = \alpha_n(\omega)$ when $\omega \in U_i$.

A morphism $f\colon (X, M_X) \to (Y, M_Y)$ between quasi-Borel spaces is function such that for all $\alpha \in M_X$, $(f \circ \alpha) \in M_Y$.

Proposition 5.5 ([Heunen et al. 2017]). *Quasi-Borel spaces and morphisms form a category* **Qbs** *that is cartesian closed. Standard Borel spaces* (X, Σ_X) *fully embed in* **Qbs**, *taking* M_X *to be the measurable functions.*

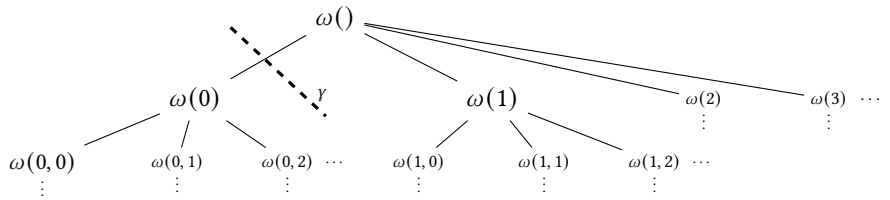
5.4 A category of probability kernels

We now revisit the intuition about randomized functions from Section 5.2 from a more formal perspective. The key idea is that X and Y there should be regarded as quasi-Borel spaces and the functions f, g as quasi-Borel functions. This allows us to equate randomized functions up-to equivalence of measures, giving an affine symmetric monoidal category.

5.4.1 Basic probability space. We now fix some basic ingredients:

- a standard Borel space (Ω, Σ_Ω) with a probability measure μ on it;
- a measure-preserving function
$$\gamma : (\Omega, \mu) \to (\Omega \times \Omega, \mu \otimes \mu).$$
i.e. for all $f\colon \Omega \times \Omega \to \mathbb{R}$, $\int f(\gamma(\omega))\, \mu(\mathrm{d}\omega) = \int \int f(\omega_1, \omega_2)\, \mu(\mathrm{d}\omega_2)\, \mu(\mathrm{d}\omega_1)$;
- a chosen uniformly distributed random variable $v\colon \Omega \to [0, 1]$.

A canonical example is to let $\Omega = [0, 1]^{\mathbb{N}^*}$, where $\mathbb{N}^*$ is the set of finite lists of natural numbers, and let
$$\gamma(\omega) = \big(\lambda(i_1, \ldots, i_n).\, \omega(0, i_1, \ldots, i_n),\ \lambda(i_1, \ldots, i_n).\, \omega(i_1 + 1, \ldots, i_n)\big)$$
In fact, this γ is an isomorphism. For an intuition, recall that a list of natural numbers describes a path to a node in the tree that is infinitely deep and infinitely wide (sometimes called a 'rose tree'). So each $\omega \in \Omega$ is an infinitely wide and deep tree where every node is annotated with a real number, and γ splits the tree as indicated by the dotted line:



Our probability measure μ on this choice of Ω is the countably-infinite product measure of the uniform distribution, given by the Kolmogorov extension theorem. For each path $(i_1, \ldots, i_n) \in \mathbb{N}^*$, the projection function gives a random variable $\Omega \to [0, 1]$, which is uniformly distributed, and these are all independent. In particular, the empty path gives $v : \Omega \to [0, 1]$, with $v(\omega) = \omega()$.

5.4.2 Probability kernels. We return to the question of when two randomized functions should be equated (§5.2). Suppose that Ω is equipped with a probability measure, as in Section 5.4.1. Although a quasi-Borel space Y is not (a priori) a measurable space, we still have a construction like a 'push-forward measure' along any morphism $f : \Omega \to Y$. By this we mean that for any morphism $h : Y \to \mathbb{R}$, we can define its 'expected value' to be $\int h(f(\omega))\, \mu(\mathrm{d}\omega)$. This is a Lebesgue integral because the composite function $hf : \Omega \to \mathbb{R}$ is a morphism, and Ω and $\mathbb{R}$ are standard Borel spaces, so hf is measurable (Prop. 5.5). In traditional measure theory, two measures inducing the same expectation operation must be equal. We use this intuition to formulate when randomized functions should be equated in this setting.

Definition 5.6. Let X and Y be quasi-Borel spaces. A *probability kernel* $f : X \rightsquigarrow Y$ is a quasi-Borel function $f : X \times \Omega \to Y$. We consider the equivalence relation on probability kernels that is determined by

$$f \sim g \text{ if for all } x \in X \text{ and all morphisms } h : Y \to \mathbb{R}, \int h(f(x,\omega))\,\mu(\mathrm{d}\omega) = \int h(g(x,\omega))\,\mu(\mathrm{d}\omega). \tag{9}$$

We can perform various constructions on probability kernels:

- There is a probability kernel $1 \rightsquigarrow \mathbb{R}$ which describes the uniform distribution on the unit interval $[0,1]$, coming from $\upsilon : \Omega \to [0,1]$.
- For any X, the identity probability kernel $X \rightsquigarrow X$ is the projection function $X \times \Omega \to X$.
- We *compose* two probability kernels $f : X \rightsquigarrow Y$, $g : Y \rightsquigarrow Z$, obtaining a probability kernel $gf : X \rightsquigarrow Z$ given by:
$$X \times \Omega \xrightarrow{X \times Y} X \times \Omega \times \Omega \xrightarrow{f \times \Omega} Y \times \Omega \xrightarrow{g} Z$$
- We *tensor* two probability kernels $f : A \rightsquigarrow B$, $g : X \rightsquigarrow Y$, obtaining a probability kernel $f \otimes g : (A \times X) \rightsquigarrow (B \times Y)$ given by:
$$A \times X \times \Omega \xrightarrow{A \times X \times Y} A \times X \times \Omega \times \Omega \xrightarrow{\cong} A \times \Omega \times X \times \Omega \xrightarrow{f \times g} B \times Y$$

Proposition 5.7. *Probability kernels modulo equivalence (9) form a monoidal category* **ProbKer***: that is, composition and tensor are associative and unital up to equivalence, the interchange law is satisfied up to equivalence, and the operations on probability kernels respect equivalence relations.*

We can regard any quasi-Borel function $f : X \to Y$ as a probability kernel $(X \times \Omega \xrightarrow{\mathtt{fst}} X \xrightarrow{f} Y)$; this induces an identity-on-objects functor **Qbs** $\to$ **ProbKer**.

Proposition 5.8 (see [Heunen et al. 2017]). *The inclusion functor* **Qbs** $\to$ **ProbKer** *has a right adjoint. That is, the functions* $(\Omega \to X)$ *modulo equivalence (9) form an affine commutative monad on the category of quasi-Borel spaces.*

5.5 A category of measure kernels

We now turn to unnormalized measures. The notion of probability kernel on quasi-Borel spaces accounts for the basic notion of pushing forward a probability measure along a function. The other key method for building probability measures, and measures generally, is using *densities* or *weights*. For example, the density of the beta distribution, $6x(1-x)$, defines a measure on the unit interval. Densities can also construct unnormalized measures: starting from the standard normal distribution on $\mathbb{R}$, the weight $(\sqrt{2\pi})e^{\frac{1}{2}x^2}$ defines the Lebesgue measure on $\mathbb{R}$, which assigns to each open interval its length (see also [Staton 2020]).

A parameterized measure, i.e. a measure kernel, will thus be a probability kernel together with a weight function. As motivated in Section 2, this matches the two operations forming measures in the metalanguage, `sample` and `score`.

Definition 5.9. A *measure kernel* $(f, \ell) : X \rightsquigarrow Y$ is a pair of quasi-Borel functions, $f : X \times \Omega \to Y$, $\ell : X \times \Omega \to [0,\infty]$. We consider the equivalence relation on probability kernels that is determined by $(f,\ell) \sim (f',\ell')$ if for all $x \in X$ and all morphisms $g : Y \to \mathbb{R}$,

$$\int \ell(x,\omega) \cdot g(f(x,\omega))\,\mu(\mathrm{d}\omega) = \int \ell'(x,\omega) \cdot g(f'(x,\omega))\,\mu(\mathrm{d}\omega).$$

We can perform various constructions on measure kernels too.

- Any probability kernel $X \rightsquigarrow Y$ can be regarded as a measure kernel with constant weight 1.

- Any morphism $w : X \to \mathbb{R}$ can be regarded as a measure kernel $X \rightsquigarrow 1$ onto the one point space.
- We *compose* measure kernels by composing the probability kernels and multiplying the weights.
- We *tensor* measure kernels by tensoring the probability kernels and multiplying the weights.
- We can regard any quasi-Borel function $X \to Y$ as a measure kernel $X \rightsquigarrow Y$, with weight constant 1; this induces an identity-on-objects functor **Qbs** $\to$ **MeasKer**.

Proposition 5.10 ([Ścibior et al. 2018], §4.3.3). *Measure kernels modulo equivalence form a monoidal category. The inclusion functor* **Qbs** $\to$ **MeasKer** *has a right adjoint, and so the functions* $\Omega \to (X \times \mathbb{R})$ *modulo equivalence form a commutative monad on the category of quasi-Borel spaces.*

5.6 Summary, and alternative approaches and variations

5.6.1 Summary. The category of quasi-Borel spaces provides a model of the metalanguage from Section 2, as follows. We understand `RealNum` as the quasi-Borel space of real numbers, $\mathbb{R}$.

- The `Prob` monad is induced by the category of probability kernels, and is affine and commutative, by Proposition 5.8.
- The `Meas` monad is induced by the category of measure kernels, and is commutative, by Proposition 5.10.
- The morphism `sample :: Prob a → Meas a` is induced by the inclusion of the category of probability kernels into the category of measure kernels, taking constant weights ($\ell = 1$).
- The morphism `score :: RealNum → Meas ()` is induced by putting $\ell : \mathbb{R} \times \Omega \to [0, \infty]$ as the absolute value of the left projection.

We also have the concrete distributions used in the examples in Section 3. All distributions on $\mathbb{R}$ arise as pushforwards of the uniform distribution on $[0, 1]$, so they are present; the morphism `iid :: Prob RealNum → Prob (Stream RealNum)` is defined by the Kolmogorov extension theorem.

As an aside we note another clue that a distinction between `Prob` and `Meas` is needed. If we had a quasi-Borel morphism `iidMeas :: Meas RealNum → Meas (Stream RealNum)` in this measure-theoretic situation, it could be used to build an infinite-dimensional Lebesgue measure on $\mathbb{R}^{\mathbb{N}}$, which is well-known to be problematic [Baker 1991]. By contrast an infinite-dimensional uniform probability distribution is straightforward, by Kolmogorov extension, and is very useful.

5.6.2 Categories of measure kernels. The probability measures on a measurable space X form a affine commutative monad on the category of measurable spaces and measure-preserving maps, called the Giry monad [Giry 1982]. Moreover, the s-finite measure kernels between measurable spaces form a monoidal category [Staton 2017]. This more traditional measure-theoretic foundation forms a good intuition for our metalanguage (§2) (see also [Kozen 1981]), but unlike quasi-Borel spaces, it cannot actually model the metalanguage, for two reasons: first, the category is not cartesian closed, so it does not support all function types [Aumann 1961]; second, it is not known whether the s-finite measure kernels form a Kleisli category. These issues with the category of measurable spaces led us to use quasi-Borel spaces, where they vanish, and so we use this instead as a semantic basis.

5.6.3 Alternative representations of randomized functions. We note a different notion of randomized function, where the function is equipped with a parameter space (following e.g. [Shiebler 2020]; see also [Lew et al. 2020; Ścibior et al. 2018]). Let us briefly define a *para-randomized* function between sets X and Y to be a pair (Ω, f), where Ω is a standard probability space and $f : X \times \Omega \to Y$ is a function. Unlike with our randomized functions, the seed space is not fixed and is part of the data

for a para-randomized function. Composition is of the form

$$f\colon X \times \Omega_1 \to Y \qquad g\colon Y \times \Omega_2 \to Z \qquad (g \circ f)\colon X \times (\Omega_1 \times \Omega_2) \to Z$$

with $(g \circ f)(x, (\omega_1, \omega_2)) = g(f(x, \omega_1), \omega_2)$. This formulation is convenient when a function has a natural parameter space of fixed dimension such as $\mathbb{R}^3$. Then composing $(f, \mathbb{R}^m)$ and $(g, \mathbb{R}^n)$ yields $(g \circ f, \mathbb{R}^{m+n})$. This is reasonable for certain simple probabilistic programs, but in this article we are especially interested in the situation where the parameter spaces are not so simple. For example, in §3.2 we compose each point of a Poisson point process, of infinite dimension, with a random linear function; it is not so clear how to manage this straightforwardly by combining dimensions.

5.6.4 Domain theoretic models and recursion. The equational definitions in Sections 2–4 all have solutions in quasi-Borel spaces. For general higher order recursion, one can extend quasi-Borel spaces by placing a cpo structure on the carrier, and imposing compatibility conditions, see [Vákár et al. 2019]. We omit the details, which are largely orthogonal.

A different approach would be to use the recent work of [Goubault-Larrecq et al. 2021] to build an entirely domain theoretic model. There are also other potential ways to interpret laziness in probabilistic models of linear logic [Ehrhard et al. 2018; Lago and Hoshino 2019; Maraist et al. 1999]. Here we have stuck with quasi-Borel spaces because they also connect to our implementation (§5.7).

5.6.5 Open questions. We briefly remark that although the category of quasi-Borel spaces accommodates all the examples from Section 3, the abstraction and generality of the metalanguage opens up challenges. For example, we do have `iid` for spaces that are standard Borel, but Kolmogorov extension beyond that is an open question; semantic models of stochastic memoization for uncountable domain spaces (e.g. white noise) is an open question; semantic models with Gaussian processes with discontinuous kernels is an open question. These are all well-known challenges for traditional measure-theoretic probability, but they can now be phrased in precise terms through the metalanguage. (Curiously, none of these things are at all problematic in our implementation (§5.7).)

5.7 Haskell implementation of the probability and measures monads

In Sections 5.2–5.5 we gave a semantic model of the metalanguage (§2). This leads directly to our implementation in the LazyPPL library [Dash et al. 2022b]. For the probability space, we put $\Omega =$ `Tree` and $\gamma =$ `splitTree`:

```
data Tree = Tree Double [Tree]
```

```
splitTree :: Tree → (Tree , Tree)
splitTree (Tree r (t : ts)) = (t , Tree r ts)
```

A probability distribution over `a` is a function `Tree → a`.

```
newtype Prob a = Prob (Tree → a)
return a = Prob $ const a
```

```
uniform :: Prob Double
uniform = Prob $ (Tree r _) → r
```

```
(Prob m) >>= f = Prob $ \g → let {(g1,g2) = splitTree g; (Prob m') = f (m g1)} in m' g2
```

Note that although the type looks like the reader monad, the bind is different. A similar bind is used in QuickCheck [Claessen and Hughes 2000, §6.4], although we are not aware of a semantic analysis in the literature.

We implement the measures monad using the writer monad transformer. Because weights multiply, they often become very small, and so we use log numbers.

```
newtype Meas a = Meas (WriterT (Product (Log Double)) Prob a)
```

```
sample :: Prob a → Meas a
sample p = Meas $ lift p
```

```
score :: Double → Meas ()
score r = Meas $ tell $ Product $ (Exp . log) r
```

We provide several inference methods in [Dash et al. 2022b]. A simple reference method is a likelihood-weighted importance sampler, following e.g. [van de Meent et al. 2018, §4.1], which is just 20 lines of Haskell. This has type `lwis :: Int → Meas a → IO [a]`. Running `(lwis n m)` produces a stream of samples from the unnormalized measure `m`. As $n \to \infty$, the stream of samples converges to a stream of iid samples from the normalized probability distribution corresponding to `m`. But in practice, for feasible n, it is not usually very accurate, and so we look at a better algorithm in Section 6.

6 A NEW METROPOLIS-HASTINGS KERNEL FOR LAZINESS

In Section 5 we showed that a closed program in the metalanguage (§2) of type `Meas` X induces a pair of functions $\mathbb{R} \xleftarrow{\ell} \Omega \xrightarrow{f} X$, where Ω is regarded with a basic probability measure p, and ℓ is measurable. Here ℓ is regarded as a density for an unnormalized distribution on Ω, which is then to be pushed forward to X, which is the space of interest. We now describe a new Markov-Chain Monte Carlo inference algorithm that works for programs in the metalanguage under this semantics, which works well for the examples we have considered here (§3,4).

Notice that there are four measures of interest:

- The basic probability measure μ on Ω;
- The unnormalized measure μ_ℓ on Ω, induced by regarding ℓ as a density. Formally, $\mu_\ell(U) = \int_\Omega [\omega \in U] \cdot \ell(\omega)\, \mu(\mathrm{d}\omega)$. This could be written in the metalanguage as

$$\texttt{do } \{\omega \leftarrow \texttt{sample } \mu;\ \texttt{score } (\ell\, \omega);\ \texttt{return } \omega\} :: \texttt{Meas } \Omega.$$

  (Here, and throughout Section 6.1 and the proof of Thm. 6.2, we are using the metalanguage to discuss and manipulate semantic measures – these are not necessarily programs to be run directly, by contrast with Section 3.)
- The *normalized* form of the measure μ_ℓ, $\frac{\mu_\ell}{\mu_\ell(\Omega)}$, which is a probability measure, assuming $\mu_\ell(\Omega) \in (0, \infty)$.
- The pushforward probability measure on X, $f^*(\frac{\mu_\ell}{\mu_\ell(\Omega)})$. This could be written

$$\texttt{do } \{\omega \leftarrow \texttt{sample } \mu;\ \texttt{score } (\ell\, \omega);\ \texttt{score } (1/(\mu_\ell\ \Omega))\ ;\ \texttt{return } (f\, \omega)\} :: \texttt{Meas } X.$$

The challenge is that the normalizing constant $\mu_\ell(\Omega)$ is typically difficult to calculate. The Markov-Chain Monte Carlo simulation algorithms provide a sampling procedure for $\frac{\mu_\ell}{\mu_\ell(\Omega)}$ on Ω, without explicitly calculating $\mu_\ell(\Omega)$. They are best described as algorithms over Ω, rather than X, although we can push-forward the samples to X at the last minute.

6.1 Proposal kernels in general

The key ingredient for a Metropolis-Hastings algorithm is a 'proposal' Markov kernel. This is a function $k : \Omega \times \Sigma_\Omega \to [0, 1]$ such that each $k(\omega, -)$ is a probability measure and each $k(-, U)$ is measurable. We follow the analysis of proposal kernels from [Geyer 2011; Green 1995].

The proposal kernel k does not directly capture the probability measure $\frac{\mu_\ell}{\mu_\ell(\Omega)}$. Rather, it induces another kernel, which works by first proposing changes (using k) and then either accepting or rejecting them (§6.3). This depends on an 'acceptance ratio' which exists as long as k satisfies the Green property, that we define now.

Given a Markov kernel k we can form an unnormalized kernel by composing it with the unnormalized measure μ_ℓ. This gives two measures m, m_{rev} on $\Omega \times \Omega$:

$$m(U) = \int_\Omega \int_\Omega [(\omega_1, \omega_2) \in U] \cdot \ell(\omega_1)\, k(\omega_1, \mathrm{d}\omega_2)\, \mu(\mathrm{d}\omega_1)$$
$$m_{rev}(U) = \int_\Omega \int_\Omega [(\omega_2, \omega_1) \in U] \cdot \ell(\omega_1)\, k(\omega_1, \mathrm{d}\omega_2)\, \mu(\mathrm{d}\omega_1)$$

These can be described in programming terms as

```
m     = do {ω1 ← sample μ ; ω2 ← sample (k ω1) ; score (l ω1) ; return (ω1,ω2)}
m_rev = do {ω1 ← sample μ ; ω2 ← sample (k ω1) ; score (l ω1) ; return (ω2,ω1)}
```

Definition 6.1 ([Green 1995]). We say that a kernel k is *Green* with respect to ℓ and μ if m_{rev} is absolutely continuous with respect to m. This means that there exists a 'ratio' $r : \Omega \times \Omega \to \mathbb{R}$ (the 'Radon-Nikodym derivative') such that

$$\int [(\omega_1, \omega_2) \in U] \cdot r(\omega_1, \omega_2) \cdot \ell(\omega_1)\, k(\omega_1, \mathrm{d}\omega_2)\, \mu(\mathrm{d}\omega_1) = \int [(\omega_2, \omega_1) \in U] \cdot \ell(\omega_1)\, k(\omega_1, \mathrm{d}\omega_2)\, \mu(\mathrm{d}\omega_1)$$

or in programming terms

```
do {ω1 ← sample μ ; ω2 ← sample (k ω1); score (ℓ ω1); score (r ω1 ω2); return (ω1,ω2)}
= do {ω1 ← sample μ ; ω2 ← sample (k ω1); score (ℓ ω1); return (ω2,ω1)}
```

6.2 A new proposal kernel for lazy rose trees

Recall our choice of Ω is rose trees: infinitely deep and infinitely wide trees labelled from $[0, 1]$, with the basic probability measure μ giving the uniform distribution to all nodes. We consider a new proposal kernel, parameterized by a probability $p \in [0, 1]$:

- for every node, toss a coin with bias p; if heads, resample from the uniform distribution on $[0, 1]$, if tails, leave it alone.

This requires an infinite number of changes, but since probability is treated lazily, there is no problem in practice.

```
mutateTree :: RealNum → Tree → Prob Tree
mutateTree p (Tree a ts) = do b ← bernoulli p
                              a' ← uniform
                              ts' ← mapM (mutateTree p) ts
                              return $ Tree (if b then a' else a) ts'
```

This can be defined measure-theoretically using Kolmogorov's extension theorem.

Theorem 6.2. *The kernel $k : \Omega \times \Sigma_\Omega \to [0, 1]$ given by* `(mutateTree p)` *is Green, and the ratio is $r(\omega_1, \omega_2) = \frac{\ell(\omega_2)}{\ell(\omega_1)}$.*

Proof notes. Notice that k is reversible with respect to μ in that

```
do {ω1 ← μ; ω2 ← k ω1; return (ω1,ω2)} = do {ω1 ← μ; ω2 ← k ω1; return (ω2,ω1)}
```

This can be deduced from Kolmogorov's extension theorem, by proving it for finite projections. Therefore the given r is indeed a ratio, since

```
do {ω1 ← sample μ; ω2 ← sample (k ω1); score (ℓ ω1); score (r ω1 ω2); return (ω1,ω2)}
= do {ω1 ← sample μ; ω2 ← sample (k ω1); score (ℓ ω2); return (ω1,ω2)}
= do {ω1 ← sample μ; ω2 ← sample (k ω1); score (ℓ ω1); return (ω2,ω1)}
```

as required, where the second step uses the reversibility of k with respect to μ. □

Technical note. Our k is reversible in the given sense, and this appears to be a 'Metropolis ratio'. But because our space Ω is infinite-dimensional, the traditional density-based analysis of Metropolis does not apply, whereas this more general approach by Green does.

6.3 The Metropolis-Hastings-Green Markov Chain

Let $k : \Omega \times \Sigma_\Omega \to [0,1]$ be a Green Markov kernel with ratio $r : \Omega \times \Omega \to [0,1]$. The *Metropolis-Hastings-Green kernel* $k_{MHG} : \Omega \times \Sigma_\Omega \to [0,1]$ is now given by *proposing* a new ω_2 via $k(\omega_1, -)$, and accepting or rejecting the proposal according to $\min(1, r(\omega_1, \omega_2))$. Either way, we produce something, either the new ω_2 or the old ω_1.

```
k_MHG :: Ω → Prob Ω
k_MHG ω₁ =
 do {ω₂ ← k ω₁ ; b ← bernoulli $ min 1 (r ω₁ ω₂) ; if b then return ω₂ else return ω₁}
```

We can then construct a Markov chain with transitions given by k_{MHG}. The key result (e.g. [Geyer 2011; Green 1995]) is that when k is well-behaved, the states of this Markov chain approximate the posterior distribution. Theorem 6.3 says this formally. Recall that a probability measure ν on Ω is a stationary distribution for a kernel $k : \Omega \times \Sigma_\Omega \to [0,1]$ if

$$\int_\Omega k(\omega, U) \cdot \nu(\mathrm{d}\omega) = \nu(U).$$

We say that k is irreducible with respect to a probability measure ξ if for every $\omega \in \Omega$ and for every $U \in \Sigma_\Omega$ such that $\xi(U) > 0$, there exists $n \in \mathbb{N}$ such that $k^n(\omega, U) > 0$. Informally, irreducibility means that the Markov chain will reach any set of positive measure in finite time.

Theorem 6.3 (Metropolis-Hastings-Green). *For any Green kernel k, the induced kernel k_{MHG} has a stationary distribution, which is the normalized probability measure $\frac{\mu_\ell}{\mu_\ell(\Omega)}$ on Ω. If k_{MHG} is irreducible with respect to $\frac{\mu_\ell}{\mu_\ell(\Omega)}$ then the stationary distribution is unique.*

We can therefore use the Metropolis-Hastings-Green kernel as a method for sampling from the normalized probability measure.

Proposition 6.4. *For the* `mutateTree` *kernel (§6.2) with $p = 1$, k_{MHG} is irreducible for $\frac{\mu_\ell}{\mu_\ell(\Omega)}$.*

Proof note. Here $n = 1$ suffices. □

We recall that correctness of a similar 'all-sites' Metropolis-Hastings scheme for probabilistic programming was proved in [Borgstrom et al. 2016], albeit for a non-lazy language.

There remains a concern that k_{MHG} is not irreducible for $p < 1$. Indeed, in that situation, the set $U = \{\omega' \mid \forall i.\, \omega_i \neq \omega'_i\}$ is not reachable from ω, even though U typically has measure 1. More informally, although every node has a chance of being changed, there will almost surely exist a node that is not changed. In practice, (`mutateTree p`) alone appears to be fine, because any finite collection of samples will only invoke a finite number of nodes anyway. We return to this point in Section 7.2.

6.4 Summary and example

In summary, we have a procedure for sampling from the distribution described by a program in the metalanguage (§2), by using the Metropolis-Hastings-Green kernel (§6.3) associated to the Green Markov kernel (`mutateTree p`) (§6.2). Each step of the algorithm provides a sample from the

measure $\frac{\mu_\ell}{\mu_\ell(\Omega)}$ on Ω, and we can push-forward this sample along $f\colon \Omega \to X$ to obtain a sample from the measure described by the program.

To illustrate, we recall the simple linear regression model (§2.1). Although we are using an infinite tree, only two samples will be used, for the slope `a` and intercept `b`. If we use our kernel with $p = 0.5$, at each step, *one* of the following steps will happen, each with probability 0.25.

- We will change neither `a` nor `b`. (This is a wasted step.)
- We try to change the slope `a` but keep the intercept `b` the same. This is useful if they are independent.
- We try to change the intercept `b` but keep the slope `a` the same. Again, this is useful if they are independent.
- We try to change both the slope `a` and the intercept `b`. This is sometimes called 'multisite' inference, and is useful if they are correlated.

As is always the case with general purpose methods, it is non-optimal if the independence and correlations are known. But our algorithm serves well where they are not known, and moreover works perfectly well with the lazy structures used in the probability monad.

7 MIXED KERNELS AND SINGLE-SITE METROPOLIS-HASTINGS

There are many possible variations on the generic inference algorithm in Section 6. In this section we consider the mixture of kernels — randomly choosing between different kernels at each step. This has at least two useful applications:

- Mixing (`mutateTree p`) with (`mutateTree 1`), which ensures a unique stationary distribution, and intuitively allows the entire tree to be reset sometimes, which can be useful for exploring multimodal distributions (§7.2);
- A single-site proposal kernel, where we mutate exactly one node in each step, following Wingate et al. [Wingate et al. 2011] (§7.3).

7.1 State-dependent mixing in general

We consider the following general method for mixing Green kernels (Def. 6.1), which is perhaps implicit in [Geyer 2011; Green 1995]. Let $k_i : \Omega \times \Sigma_\Omega \to [0,1]$ be a countable family of Markov kernels (§6.1), and let $c\colon \Omega \times \mathbb{N} \to [0,1]$ be a parameterized probability distribution function over $\mathbb{N}$, i.e. for all $\omega \in \Omega$, $\sum_{i=1}^{\infty} c(\omega, i) = 1$. Let k be the mixed kernel

$$k(\omega, U) = \textstyle\sum_{i=1}^{\infty} c(\omega, i) \cdot k_i(\omega, U). \tag{10}$$

Suppose that each k_i is a Green kernel with respect to μ, with ratio $r_i : \Omega \times \Omega \to [0,\infty]$ (Def. 6.1). Suppose that we can always detect which kernel was used, i.e. there is a function $e : \Omega \times \Omega \to \mathbb{N}$ such that `do {`$\omega \leftarrow \mu$`; `$\omega' \leftarrow k_i\,\omega$`; return (`$\omega, \omega', e(\omega,\omega')$`)} = do {`$\omega \leftarrow \mu$`; `$\omega' \leftarrow k_i\,\omega$`; return (`$\omega, \omega', i$`)}`.

Theorem 7.1. *The kernel k (10) is a Green kernel with respect to μ, with ratio $r\colon \Omega \times \Omega \to [0,\infty]$*

$$r(\omega, \omega') = r_i(\omega, \omega') \cdot \frac{c(\omega', i)}{c(\omega, i)} \quad \textit{where } i = e(\omega, \omega').$$

7.2 Application 1: mixing for unique stationary distributions

A special case of Theorem 7.1 is independent mixing, which is a simple way of combining kernels, and a way of building irreducible MHG kernels (i.e. with unique stationary distributions, §6.3):

Proposition 7.2. *If k and k' are Green kernels, and k induces an irreducible MHG kernel, then for $r \in (0,1)$, the kernel $(r \cdot k + (1-r) \cdot k')$ again induces an irreducible MHG kernel.*

Proof notes. The kernel $(r \cdot k + (1-r) \cdot k')$ is Green by Theorem 7.1. For irreducibility, consider ω and U; since k is irreducible there is n such that $k^n(\omega, U) > 0$, so $(r \cdot k + (1-r) \cdot k')^n(\omega, U) > r^n \cdot k^n(\omega, U) > 0$. □

In particular, completing the discussion from Section 6.3, (`mutateTree p`) might not induce an irreducible kernel, but if an irreducible kernel is desired then we can instead start from the mixed kernel $(r \cdot$ (`mutateTree 1`$) + (1-r) \cdot$ (`mutateTree p`$))$, using Prop. 6.4.

7.3 Application 2: single-site proposal kernel for lazy rose trees

The single-site Metropolis-Hastings proposal kernel is a popular generic kernel for probabilistic programming ([Wingate et al. 2011],[van de Meent et al. 2018, §4.2.1]). In a finite-dimensional situation, the idea is to randomly pick one dimension and change it, leaving the other dimensions unchanged.

The subtlety here is that in a lazy program, such as the examples in Sections 3 and 4, the number of dimensions is unbounded, and so it is a priori impossible to pick one dimension uniformly at random. Nonetheless, we now use Theorem 7.1 to show that there is actually a well-behaved analogue of this kernel that is relevant where there is lazy structure. The idea is to inspect the dimensions that are actually used in a given computation.

High-level view. Recall the representation of probabilistic programs developed in Sections 5 and 6, with Ω the infinite rose trees, and weight function $\ell : \Omega \to [0, \infty]$, and an outcome $\Omega \to X$. We describe the single-site proposal kernel at this level. To do this, we instantiate state-dependent mixing as follows. We work up-to a bijection between natural numbers and finite paths through the rose tree, which are countably infinite.

- For each path i through the rose tree, let k_i be the kernel that randomly changes node i and leaves the others unchanged. This is a Green kernel with ratio $\ell(\omega')/\ell(\omega)$.
- If ω and ω' differ by only one node, then let $e(\omega, \omega')$ return the path to this node.
- For any given tree ω, we define $c(\omega, i)$ as follows. First, if ℓ and f are defined by programs that terminate, then we calculate the necessarily finite set of nodes $S_\omega = \{i_1 \dots i_n\}$ that are actually inspected in evaluating $\ell(\omega)$ and $f(\omega)$. We then pick one at random, i.e. let $c(\omega, i) = \frac{1}{|S_\omega|}$ if $i \in S_\omega$, and $c(\omega, i) = 0$ otherwise.
- Following Theorem 7.1, we can calculate the Green ratio as $\frac{\ell(\omega') \cdot |S_\omega|}{\ell(\omega) \cdot |S_{\omega'}|}$.

Implementation details. Lazy evaluation is the sole reason why we are even able to consider 'the set of nodes in the tree that have been evaluated' in any given run of our probabilistic program, and it ensures that no irrelevant sites are present in that set (i.e. those sites which do not affect the outcome of the result of that run). In our implementation of the single-side proposal kernel, we go under the hood and inspect system memory from within Haskell to calculate this set of sites S_ω. The `ghc-heap` module exposes the parts of the tree that have been evaluated to weak-head normal form and parts still untouched (present in memory as thunks), and using this we can safely inspect the runtime evaluation state of our random tree without forcing any further computation on it.

A note on performance. General purpose Metropolis-Hastings methods cannot be expected to out-perform hand-crafted methods, or more specialized Monte Carlo algorithms [van de Meent et al. 2018], and so we have not carried out a detailed performance evaluation, but the performance was perfectly adequate for the examples in Sections 3 and 4. Our main intention for this section was to demonstrate that the popular single-site proposal kernels can still be used in the lazy setting. That said, we can make some brief anecdotal remarks: we found that sometimes, where there are very many independent sites, such as in cluster assignment, the single-site method will perform

better, whereas in many other situations the method of Section 6 performs well; in multimodal situations, such as mixture models, we found it is beneficial to use Section 7.2, allowing a small chance of resetting the tree and exploring a totally different region.

Our illustrations in Figures 3 and 5 (see [Dash et al. 2022b] for more) show that these inference methods produce reasonable results in practice. We also remark briefly on the time/space usage of the current implementation in Haskell. The three regression illustrations in Figure 3 took around 2s, 18s and 19s respectively (AWS EC2 t3.large). For instance, in Figure 3(b) we ran 10^6 steps of the Markov chain, thinning the samples for legibility. Although these timings are fine, in our experience, memory usage can be a concern with larger models in the current implementation. For example, we tested a program induction example which is highly multimodal (see [Dash et al. 2022b]), and it typically needs more than 8GB RAM to produce 10^6 samples.

8 RELATED WORK ON LAZINESS AND PRACTICAL SYNTHETIC PROBABILITY

Our aim in this work is to study the power of types and laziness as a practical synthetic measure theory. Our work is inspired by many other developments on the practical front.

8.1 Laziness in probabilistic programming languages

The Church project [Goodman et al. 2008] is a major inspiration for our work. Although Church is an eager language, it has a primitive memoization construct (c.f. §4.2). This leads to a programming style for lazy behaviour: instead of writing `do {x ← t; y ← u; z ← v; ...}` and expecting lazy evaluation, one can write (roughly) `f ← memoize(\i → case i of {1 → t; 2 → u; 3 → v}); ...` with eager evaluation, and use `f 1`, `f 2`, `f 3` in place of `x`, `y`, `z` respectively. Although this is an unusual programming style, it is usable nonetheless. Since Church is untyped, the precise connection with our metalanguage (§2) is unclear, and the semantics seems slightly different. But the connection with non-parametric statistics is heavily emphasized, for example in the analysis of stick-breaking [Goodman et al. 2008; Roy et al. 2008] and exchangeable primitives [Wu 2013]. In summary, from a bird's eye view, our metalanguage (§2) and implementation [Dash et al. 2022b] form a variation of Church with more idiomatic laziness, a type system and a semantics.

Languages such as Anglican [Tolpin et al. 2016], WebPPL [Goodman and Stuhlmüller 2014], BayesDB [Saad and Mansinghka 2017] and Turing [Bloem-Reddy et al. 2017] follow within the tradition of Church, exploring ideas from non-parametric statistics further. The Birch language [Murray and Schön 2018] is class-based, transpiling to C++ (and so the connection to Section 2 is unclear), but Birch heavily uses laziness and advanced control flow manipulations in its inference methods [Murray et al. 2018; Murray 2020].

Beyond these examples, laziness has been explored in various aspects of probabilistic programming, dating back at least as far as the pioneering work by Koller et al. [Koller et al. 1997], and more recently in the work on lazy factored inference in Figaro [Pfeffer et al. 2015], and efficient implementation in delimited continuations through Hansei ([Kiselyov and Shan 2009], which focuses on discrete distributions).

8.2 Other probabilistic programming work using Haskell and quasi-Borel spaces

Various libraries have exploited Haskell for probabilistic programming. Hakaru [Narayanan et al. 2016; Narayanan and Shan 2020; Walia et al. 2019] provides a DSL with impressive symbolic inference methods. Stochaskell [Roberts et al. 2019] provides a DSL which compiles to Stan, Church and other back-ends. Stochaskell moreover allows a limited form of lazy lists, implemented via Church's memoization.

Our work here is most heavily inspired by MonadBayes [Ścibior et al. 2018], which is a monad-based implementation of a variety of inference combinators, also inspired by the formalism of

quasi-Borel spaces [Ścibior et al. 2018]. Originally, MonadBayes was not fully lazy: the Metropolis-Hastings simulation was based on the state monad and did not support laziness. The LazyPPL project grew out of adding laziness to MonadBayes, leading to the developments and the more natural expression of the various examples in this paper. For simplicity, we have focused here on the Metropolis-Hastings inference, but in practice it would be appropriate to adapt other MonadBayes inference combinators to the lazy setting. This was explored in a recent version of MonadBayes, using a portion of the LazyPPL implementation [Cohn-Gordon 2022; Cohn-Gordon et al. 2022].

Going beyond the inference combinators of MonadBayes, quasi-Borel spaces are also a foundation for a new dependent type system based on 'trace types' [Lew et al. 2020] (prototyped in Haskell). This provides a well-typed account of the 'programmable inference' that makes recent languages such as Gen [Cusumano-Towner et al. 2019] and Pyro [Bingham et al. 2018] so powerful in practice. Wasabaye [Nguyen et al. 2022] connects trace types to Haskell's type-level strings. A possibly fruitful direction would be to generalize the traces allowed in trace types to accommodate the laziness and rose-tree-based sample space that we use in this paper.

Further beyond our aims here, quasi-Borel spaces have also found profit in many other areas of probabilistic programming, including program logics (e.g. [Aguirre et al. 2021; Sato et al. 2019]) and functional languages for probabilistic network verification ([Vandenbroucke and Schrijvers 2020]).

8.3 Other implementations of synthetic probability theory

Finally we note two other approaches to metalanguages for categorical probability theory. The first is the EfProb library [Cho and Jacobs 2017], a python library inspired by the effectus theory foundation for probability [Cho et al. 2015]. The second is a python/F# library for exact conditioning over Gaussian-based models [Stein 2021a], inspired by categorical constructions over Markov categories [Fritz 2020; Stein and Staton 2021]. These approaches are currently focused on more refined notions of conditional probability, in contrast to our approach which is based on the measure-theoretic foundations of general purpose Monte Carlo-based inference.

9 SUMMARY

We have presented a metalanguage for lazy probabilistic programming with two monads (for probability and measure, §2) and new Metropolis-Hastings-based algorithms (§6, §7). The methods are based on recent foundations from quasi-Borel spaces and synthetic probability theory (§5).

The separation into two monads is essential for decidability reasons (Theorem 2.1, Prop. 2.2), but also yields a useful programming idiom for a variety of infinite-dimensional Bayesian models, including piecewise linear regression (§3.2), non-parametric clustering (§3.3), non-parametric feature extraction (§3.4), and Gaussian process regression (§4.1), and compositions of these. As we have shown (for instance by considering rescaling), laziness allows for compositional programming, avoiding the problem of passing around truncation bounds.

10 DATA-AVAILABILITY STATEMENT

The LazyPPL library artefact [Dash et al. 2022a; Staton 2022] is openly available at https://doi.org/10.5281/zenodo.7150943.

REFERENCES

Nathanael L. Ackerman, Jeremy Avigad, Cameron E. Freer, Daniel M. Roy, and Jason M. Rute. 2019. Algorithmic barriers to representing conditional independence. In *Proc. LICS 2019*.

Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. 2016. Exchangeable random primitives. In *Proceedings of the Workshop on Probabilistic Programming Semantics*.

A. Aguirre, G. Barthe, D. Garg, M. Gaboardi, S. Katsumata, and T. Sato. 2021. Higher-order probabilistic adversarial computations: categorical semantics and program logics. In *Proc. ICFP 2021*.

Robert J. Aumann. 1961. Borel structures for function spaces. *Illinois Journal of Mathematics* 5 (1961).

Richard Baker. 1991. Lebesgue Measure on $\mathbf{R}^\infty$. *Proc. AMS* 113, 4 (1991).

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* (2018).

B. Bloem-Reddy, E. Mathieu, A. Foster, T. Rainforth, Y. W. Teh, M. Lomeli, H. Ge, and Z. Ghahramani. 2017. Sampling and inference for discrete random probability measures in probabilistic programs. In *Proc. NeurIPS 2017 Workshop on Advances in Approximate Bayesian Inference*.

Johannes Borgstrom, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. 2016. A Lambda-Calculus Foundation for Universal Probabilistic Programming. In *Proc. ICFP 2016*.

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of statistical software* 76, 1 (2017).

K. Cho and B. Jacobs. 2017. The EfProb Library for Probabilistic Calculations. In *Proc. CALCO 2017*.

Kenta Cho and B. Jacobs. 2019. Disintegration and Bayesian inversion via string diagrams. *Math. Struct. Comput. Sci.* 29 (2019), 938–971.

Kenta Cho, Bart Jacobs, Bas Westerbaan, and Abraham Westerbaan. 2015. An Introduction to Effectus Theory. (2015). arxiv:1512.05813.

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. ICFP 2000*. 268–279.

Bob Coecke. 2014. Terminality implies non-signalling. In *Proc. QPL 2014*.

Reuben Cohn-Gordon. 2022. *Improving the probabilistic programming language Monad-Bayes.* https://www.tweag.io/blog/2022-10-18-monad-bayes-fellowship/.

R. Cohn-Gordon, A. Scibior, and Tweag team. 2022. *Monad-Bayes website.* https://monad-bayes-site.netlify.app/_site/about.html.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: a general-purpose probabilistic programming system with programmable inference. 221–236.

Andreas Damianou and Neil D. Lawrence. 2013. Deep Gaussian Processes. In *Proc. AISTATS 2013*.

Swaraj Dash, Younesse Kaddar, Hugo Paquet, and Sam Staton. 2022a. Affine monads and lazy structures for Bayesian programming. (Oct 2022). https://doi.org/10.5281/zenodo.7150943

Swaraj Dash, Younesse Kaddar, Hugo Paquet, and Sam Staton. 2022b. LazyPPL: Lazy Probabilistic Programming Library. https://lazyppl.bitbucket.io/ Code repository and web page with examples.

Swaraj Dash and Sam Staton. 2020. A Monad for Probabilistic Point Processes. In *Proceedings of the 3rd Annual International Applied Category Theory Conference 2020, ACT 2020, Cambridge, USA, 6-10th July 2020 (EPTCS, Vol. 333)*, David I. Spivak and Jamie Vicary (Eds.). 19–32. https://doi.org/10.4204/EPTCS.333.2

Thomas Ehrhard, Michele Pagani, and Christine Tasson. 2018. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. In *Proc. POPL 2018*.

T. Fritz. 2020. A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics. *Adv. Math.* 370 (2020).

T. Fritz, T. Gonda, and P. Perrone. 2021. De Finetti's Theorem in Categorical Probability. *Journal of Stochastic Analysis* 2, 4 (2021).

T. Fritz, T. Gonda, P. Perrone, and Eigil Fjeldgren Rischel. 2020. Representable Markov Categories and Comparison of Statistical Experiments in Categorical Probability. https://arxiv.org/abs/2010.07416

Tobias Fritz and Wendong Liang. 2022. Free gs-monoidal categories and free Markov categories. (April 2022). arXiv:2204.02284.

Tobias Fritz and Eigil Fjeldgren Rischel. 2020. Infinite products and zero-one laws in categorical probability. *Compositionality* 2 (Aug. 2020). Issue 3. https://doi.org/10.32408/compositionality-2-3

Charles Geyer. 2011. Introduction to Markov Chain Monte Carlo. In *Handbook of Markov Chain Monte Carlo*. Chapman Hall/CRC.

S Ghosal and A van der Vaart. 2017. *Fundamentals of non-parametric Bayesian inference.* CUP.

M. Giry. 1982. A categorical approach to probability theory. *Categorical Aspects of Topology and Analysis. Lecture Notes in Mathematics* (1982).

Noah Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: a language for generative models. *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*.

Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2020-10-15.

Jean Goubault-Larrecq, Xiaodong Jia, and Clément Théron. 2021. A Domain-Theoretic Approach to Statistical Programming Languages. arxiv:2106.16190.

Peter J Green. 1995. Reversible Jump Markov Chain Monte Carlo Computation and Bayesian Model Determination. *Biometrika* 82, 4 (1995), 711–732.

T.L. Griffiths and Z. Ghahramani. 2011. The Indian Buffet Process: An Introduction and Review. *Journal of Machine Learning Research* 12, 32 (2011), 1185–1224.

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *Proc. LICS 2017*.

Ralf Hinze. 2000. Memo Functions, Polytypically!. In *Proceedings of the 2nd Workshop on Generic Programming, Ponte de*. 17–32.

Daniel Huang, Greg Morrisett, and Bas Spitters. 2018. An Application of Computable Distributions to the Semantics of Probabilistic Programs. (2018). arxiv:1806.07966.

Bart Jacobs. 1994. Semantics of weakening and contraction. *Ann. Pure Appl. Logic* 69 (1994).

Bart Jacobs. 2011. Probabilities, distribution monads, and convex categories. *Theoret. Comput. Sci.* 412 (2011).

Alexander Kechris. 1987. *Classical Descriptive Set Theory*. Springer.

Oleg Kiselyov and Chung-chieh Shan. 2009. Embedded Probabilistic Programming. In *Proc. DSL 2009*.

Anders Kock. 1970. Monads on symmetric monoidal closed categories. *Arch. Math.* 21 (1970).

Anders Kock. 1971. Bilinearity and cartesian closed monads. *Math. Scand.* 29 (1971).

A. Kock. 2012. Commutative Monads as a Theory of Distributions. *Theory and Applications of Categories* 26, 4 (2012).

Daphne Koller, David McAllester, and Avi Pfeffer. 1997. Effective Bayesian Inference for Stochastic Programs. In *Proc. AAAI 1997*.

Dexter Kozen. 1981. Semantics of probabilistic programs. *J. Comput. System Sci.* 22 (1981), 328–350.

Ugo Dal Lago and Naohiko Hoshino. 2019. The Geometry of Bayesian Programming. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. IEEE, 1–13. https://doi.org/10.1109/LICS.2019.8785663

Paul B Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inform. Comput.* 185 (2003), 182–210.

Alex K Lew, Marco F Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K Mansinghka. 2020. Trace types and denotational semantics for sound programmable inference in probabilistic languages. In *Proc. POPL 2020*.

D. Lunn, D. Spiegelhalter, A. Thomas, and N. Best. 2009. The BUGS project: Evolution, critique and future directions. *Statistics in Medicine* 28, 25 (2009), 3049–3067.

John Maraist, Martin Odersky, David N Turner, and Philip Wadler. 1999. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theoretical Computer Science* 228, 1-2 (1999), 175–210.

Donald Michie. 1968. 'Memo' Functions and Machine Learning. *Nature* 218 (1968).

E. Moggi. 1991. Notions of computation and monads. *Information and Computation* (1991).

Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*. 1037–1046.

Lawrence M Murray. 2020. Lazy object copy as a platform for population-based probabilistic programming. (Jan 2020). arxiv:2001.05293.

L. M. Murray and B. Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* (2018).

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *Proc. FLOPS 2016*. 62–79.

Praveen Narayanan and Chung-chieh Shan. 2020. Symbolic disintegration with a variety of base measures. *ACM Transactions on Programming Languages and Systems* 42, 2 (2020).

Daniel Navarro and Thomas Griffiths. 2006. A Nonparametric Bayesian Method for Inferring Features from Similarity Judgments. In *Advances in Neural Information Processing Systems*, B. Schölkopf, J. Platt, and T. Hoffman (Eds.), Vol. 19. MIT Press. https://proceedings.neurips.cc/paper/2006/file/2ecd2bd94734e5dd392d8678bc64cdab-Paper.pdf

Minh Nguyen, Roly Perera, Meng Wang, and Nicolas Wu. 2022. Modular probabilistic models via algebraic effects. In *Proc. ICFP 2022*. To appear.

John W. Paisley, David M. Blei, and Michael I. Jordan. 2012. Stick-Breaking Beta Processes and the Poisson Process. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2012, La Palma, Canary Islands, Spain, April 21-23, 2012 (JMLR Proceedings, Vol. 22)*, Neil D. Lawrence and Mark A. Girolami (Eds.). JMLR.org, 850–858. http://proceedings.mlr.press/v22/paisley12.html

Avi Pfeffer, Brian Ruttenberg, Amy Sliva, Michael Howard, and Glenn Takata. 2015. Lazy Factored Inference for Functional Probabilistic Programming. (2015). arxiv:1509.03564.

David A. Roberts, Marcus Gallagher, and Thomas Taimre. 2019. Reversible Jump Probabilistic Programming. In *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 89)*, Kamalika Chaudhuri and Masashi Sugiyama (Eds.). PMLR, 634–643. https://proceedings.mlr.press/v89/roberts19a.html

Daniel Roy, Vikash Mansinghka, Noah Goodman, and Josh Tenenbaum. 2008. A stochastic programming perspective on nonparametric Bayes. In *Proc. Workshop on Non-Parametric Bayes*.

Daniel M Roy. 2014. The continuum-of-urns scheme, generalized beta and Indian buffet processes, and hierarchies thereof. *arXiv preprint arXiv:1501.00208* (2014).

Daniel M Roy, Nate Ackerman, Jeremy Avigad, Cameron Freer, and Jason Rute. 2013. Exchangeable graphs, conditional independence, and computably-measurable samplers. Talk at CCA 2013. http://cca-net.de/cca2013/slides/17_Daniel%20Roy.pdf

F Saad and V Mansinghka. 2017. Detecting dependencies in sparse, multivariate databases using probabilistic programming and non-parametric Bayes. In *Proc. AISTATS 2017*.

T. Sato, A. Aguirre, G. Barthe, D. Garg, M Gaboardi, and J. Hsu. 2019. Formal verification of higher-order probabilistic programs. In *Proc. POPL 2019*.

Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional programming for modular Bayesian inference. In *Proc. ICFP 2018*.

Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2018. Denotational Validation of Higher-Order Bayesian Inference. In *Proc. POPL 2018*.

D Shiebler. 2020. Categorical Stochastic Processes and Likelihood. In *Proc. ACT 2020*.

Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *Proc. ESOP 2017*.

Sam Staton. 2020. Probabilistic Programs as Measures. In *Foundations of Probabilistic Programming*. CUP.

Sam Staton. 2022. LazyPPL. https://bitbucket.org/samstaton/lazyppl/src/

Guy L. Steele Jr, Doug Lea, and Christine H Flood. 2014. Fast splittable pseudorandom number generators. In *Proc. OOPSLA 2014*.

Dario Stein. 2021a. GaussianInfer. https://github.com/damast93/GaussianInfer.

Dario Stein. 2021b. *Structural Foundations for Probabilistic Programming Languages*. Ph. D. Dissertation. University of Oxford.

Dario Stein and Sam Staton. 2021. Compositional Semantics for Probabilistic Programs with Exact Conditioning. In *Proc. LICS 2021*.

Romain Thibaux and Michael I Jordan. 2007. Hierarchical Beta Processes and the Indian Buffet Process. In *Proc. AISTATS 2007*.

Luke Tierney. 1994. Markov Chains for Exploring Posterior Distributions. *The Annals of Statistics* 22, 4 (1994), 1701–1728.

D. Tolpin, H. Yang, J. W. van de Meent, and F. Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. *Proceedings of the IFL*.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction to Probabilistic Programming. (2018). https://arxiv.org/abs/1809.10756

A Vandenbroucke and T Schrijvers. 2020. PloNK: functional probabilistic NetKAT. In *Proc. POPL 2020*.

M. Vákár, O. Kammar, and S. Staton. 2019. A domain theory for statistical probabilistic programming. In *Proc. POPL 2019*.

R. Walia, P. Narayanan, J. Carette, S. Tobin-Hochstadt, and C-c Shan. 2019. From high-level inference algorithms to efficient code. In *Proc. ICFP 2019*.

David Wingate, Andreas Stuhlmueller, and Noah Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proc. AISTATS 2011*.

Frank D. Wood, Cédric Archambeau, Jan Gasthaus, Lancelot James, and Yee Whye Teh. 2009. A stochastic memoizer for sequence data. In *Proc. ICML 2009*.

J Wu. 2013. *Reduced Traces and JITing in Church*. Master's thesis. MIT.

Received 2022-07-07; accepted 2022-11-07