Affine monads and lazy structures for Bayesian programming

SWARAJ DASH, YOUNESSE KADDAR, HUGO PAQUET, and SAM STATON, 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 $T(1) = 1$.) We show 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.

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 ($\S2$). The general key point is that an ‘affine’ monad can support lazy programming, but a non-affine one cannot (see $\S1.3$ 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 ($\S1.2$, $\S3$, $\S4$; [3]). We show that these are feasible by giving new Metropolis-Hastings inference algorithms ($\S1.4$, $\S6$, $\S7$) 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 [3].

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. [21, §1.1], [22, §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. [21, 68]). 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 [42], Church [24], Stan [8] and others) can often be regarded as programming languages describing unnormalized measures, that are endowed with efficient Monte Carlo samplers. Many do not allow infinite dimensional spaces (see $\S8$).

On the theoretical side, researchers have recently proposed categorical or synthetic accounts of probability theory [10, 19] and measure theory [29, 37, 60], with the aim of developing compositional structures based on commutative and affine monads and monoidal categories. There are various
Infinite process

Finite observation

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.

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 resolve 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).

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 → 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. [10, 13, 19, 20, 32, 61]). 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 \texttt{Prob} of probability measures, allowing laziness, and a non-affine
monad \texttt{Meas} of unnormalized measures, allowing the Bayesian Monte Carlo methods (§1.1). This is
the basis of our metalanguage in Section 2.

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 [3].

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. [74]). These take as an
argument a program describing an unnormalized measure, and produce a stream of samples as
output.

Traditional Monte Carlo algorithms assume a finite-dimensional state space; sometimes they can
adapt to changing dimensions (e.g. [27]). But in a purely lazy setting, 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 [3].

- 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 [74], adapted to this lazy setting, by using Haskell internals
  (\texttt{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).

\textit{Note.} We have benefited from many helpful discussions and from presenting this work in
numerous venues. Full acknowledgements in non-anonymous version.

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 ([44]) 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.
Types: \( a, b ::= \text{RealNum} | () | (a,b) | a \to b | \text{Prob} a | \text{Meas} a | \ldots \)

Terms: \( t, u ::= x | \lambda x \to t | t u | \text{do} \{ x \leftarrow t ; u \} | \text{return} t | \ldots \)

Typing judgement \( (\Gamma \vdash t :: a) \):

| \(|\| | \(|\| | \(\Gamma, x::a \vdash \lambda x \to t :: a \to b \) | \(\Gamma \vdash t :: a \to b \) | \(\Gamma \vdash u :: a \) |
| \(\Gamma, x::a, \Gamma' \vdash \lambda x \to t :: a \to b \) | \(\Gamma \vdash t u :: b \) |
| \(\Gamma \vdash t :: a \) | \(\Gamma \vdash t :: m a \) | \(\Gamma, x::a \vdash u :: m b \) |
| \(\Gamma \vdash \text{return} t :: m a \) | \(\Gamma \vdash \text{do} \{ x \leftarrow t ; u \} :: m b \) |

Typed constants:

| \(|\| | \(|\| | \(\lambda \) :: () | (,) :: a \to b | (a,b) :: (a,b) \to a | (a,b) :: a 
| \(0,1 :: \text{RealNum} \) | * :: \text{RealNum} \to \text{RealNum} \to \text{RealNum} | \ldots |
| \(\text{sample} :: \text{Prob} a \to \text{Meas} a \) | \(\text{score} :: \text{RealNum} \to \text{Meas} () \)

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

In the metalanguage, there is a distinguished type \(\text{RealNum}\), thought of as the real numbers, and there are two monads:

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

- A measures monad \(\text{Meas}\) (in red font), so that \((\text{Meas} a)\) intuitively contains unnormalized measures on \(a\).

There are two key operations:

- \(\text{sample} :: \text{Prob} a \to \text{Meas} a\), which allows us to regard a probability measure as an unnormalized measure;

- \(\text{score} :: \text{RealNum} \to \text{Meas} ()\), which provides a measure with given weight on a single point.

The \(\text{score}\) 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 \(\mu\) and standard deviation \(\sigma\), we write \(\text{score} \ (\text{normalPdf} \ \mu \ \sigma \ x)\), where \(\text{normalPdf} \ (\mu \ \sigma \ x) = e^{-(x-\mu)^2/(2\sigma^2)} / (\sigma \sqrt{2\pi})\). We use a Haskell notation for the metalanguage; we have implemented the metalanguage in Haskell (see §5.7, §6, [3]) 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

\[
a \sim \text{Normal}(0, 3) \quad b \sim \text{Normal}(0, 3) \quad f(x) = ax + b
\]

Here, the slope \(a\) and intercept \(b\) are both drawn from normal distributions.

In the metalanguage (extended with mild syntactic sugar) we can write
Affine monads and lazy structures for Bayesian programming

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).

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

Our data points are not colinear, so we do not 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:

\[ f \sim \text{Linear} \quad y_i \sim \text{Normal}(f(x_i), \sigma) \quad \text{What is } P(f|\{y_i = d_i\})? \]

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.

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

We consider 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.:

\[
\begin{align*}
\text{sample (return } t \text{)} &= \text{return } t :: \text{Meas } a \\
\text{sample (do } x ← p ; q } \text{) } &= \text{do } x ← \text{sample } p ; \text{ sample } q :: \text{Meas } b
\end{align*}
\]  

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

The measures monad should satisfy commutativity (e.g. [35]): for mx :: Meas a and my :: Meas a,

\[
\begin{align*}
\text{do } \{ x ← mx ; y ← my \ ; \text{return (x,y)} \} &= \text{do } \{ y ← my ; x ← mx ; \text{return (x,y)} \}
\end{align*}
\]  

where x ∉ freevars(my), y ∉ freevars(mx). The probability monad should satisfy commutativity, and also affinity (e.g. [31, 36]):

\[
\begin{align*}
\text{if } \text{mx :: Prob a, then: } \text{do } \{ x ← mx ; \text{return } () \} &= \text{return } ()
\end{align*}
\]  

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

\[
\begin{align*}
\text{score } 1 &= \text{return } () :: \text{Meas } () \\
\text{do } \{ \text{score } 0 ; mx \} &= \text{do } \{ \text{score } 0 ; my \} :: \text{Meas } a \\
\text{do } \{ \text{score } r ; \text{score } s \} &= \text{score } (r * s) :: \text{Meas } ()
\end{align*}
\]  

for r,s :: RealNum and mx,my :: Meas a.
Typed equational theory: we define a relation $\Gamma \vdash t = u :: a$ over typed terms $\Gamma \vdash t, u :: a$.

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

If $\Gamma \vdash t :: ()$ then $\Gamma \vdash t = () :: ()$.

Other axioms:

- $\text{do } \{ y \leftarrow \text{do } \{ x \leftarrow s ; t \} ; u \} = \text{do } \{ x \leftarrow t ; \text{return } x \} = t$
- $\text{do } \{ x \leftarrow \text{return } t ; u \} = \text{u}[t/x]$
- $\text{do } \{ x \leftarrow \text{return } t ; \text{return } t \} = \text{u}[t/x]$
- $\text{do } \{ x \leftarrow s ; \text{do } \{ y \leftarrow t ; u \} \} = \text{do } \{ x \leftarrow s ; \text{do } \{ y \leftarrow \text{do } \{ x \leftarrow s ; t \} ; u \} \} $

$\text{fst}(t,u) = t \quad \text{snd}(t,u) = u \quad t = (\text{fst } t, \text{snd } t) \quad (x \rightarrow t) \ u = t[u/x] \ u = (\lambda x \rightarrow (u x))$

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

We are not seeking a fully complete equational axiomatization, but these will prove useful sound laws for reasoning about program behaviour and laziness. In Section 5 we give a denotational model that validates these axioms, and in Section 5.7 we give details of an implementation that validates the axioms.

2.3 Affine probability monads, laziness, and iid sequences

We now show the connection between the affine monad law (3) and lazy data structures. To this end, we suppose that the metalanguage is extended with types Stream a of streams, and standard accessor methods such as

- $\text{hd} :: \text{Stream } a \rightarrow a$
- $\text{tl} :: \text{Stream } a \rightarrow \text{Stream } a$
- $\text{(:)} :: a \rightarrow \text{Stream } a \rightarrow \text{Stream } a$

together with syntactic sugar, and equations such as

- $\text{hd } (x:x:s) = x \quad \text{tl } (x:x:s) = x \quad (\text{hd } s): (\text{tl } s) = xs$

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:

- $\text{iid } :: \text{Prob } \text{RealNum} \rightarrow \text{Prob } (\text{Stream } \text{RealNum})$
- $\text{iid } p = \text{do } \{ x \leftarrow p ; xs \leftarrow \text{iid } p ; \text{return } (x : xs) \}$

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 $i_1, \ldots, i_n$ of distinct indices, the following programs of type $\text{Prob } (\text{RealNum}, \ldots, \text{RealNum})$ are equal:

- $\text{do } \{ xs \leftarrow \text{iid } p ; \text{return } (xs !! i_1,\ldots, xs !! i_n) \}$
- $\text{do } \{ x_1 \leftarrow p ; \ldots ; x_n \leftarrow p ; \text{return } (x_1, \ldots, x_n) \}$

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

- $xs \leftarrow \text{iid } p ; \text{return } (xs !! 2, xs !! 1)$

<table>
<thead>
<tr>
<th>Case</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$= x_0 \leftarrow p \ ; x_1 \leftarrow p \ ; x_2 \leftarrow p \ ; xs \leftarrow \text{iid } p ; \text{return } ((x_0:x_1:x_2:xs)!!2,(x_0:x_1:x_2:xs)!!1)$</td>
</tr>
<tr>
<td>2</td>
<td>$= x_0 \leftarrow p \ ; x_1 \leftarrow p \ ; x_2 \leftarrow p \ ; xs \leftarrow \text{iid } p ; \text{return } (x_2,x_1)$</td>
</tr>
<tr>
<td>3</td>
<td>$= x_1 \leftarrow p \ ; x_2 \leftarrow p ; \text{return } (x_2,x_1)$</td>
</tr>
<tr>
<td>4</td>
<td>$= x_1 \leftarrow p \ ; x_2 \leftarrow p ; \text{return } (x_1,x_2)$</td>
</tr>
</tbody>
</table>

In consequence, although the first program involves the recursive call to iid, in any semantics that respects the equations, it must terminate.

Using similar arguments, we can assume a more general primitive that generates a stream of results by iterating a function $b \rightarrow \text{Prob } (a, b)$. This is subject to the following equation:
We note that we could define \( \text{iid} \) by
\[
\text{iid} p = \text{unfold} (\_ \rightarrow \text{do} \{ x \leftarrow p; \text{return} \ (x, ()) \}) () .
\]

### 2.3.1 Discussion of problems with laziness in general measures monads.

We do not ask that the general measures monad, \( \text{Meas} \ a \), 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 \( \text{Meas} \) monad, as we now explain.

Suppose for a moment that we did have the equivalent of \( \text{unfold} \) and \( \text{iid} \) in the \( \text{Meas} \) monad:
\[
\text{unfoldM} :: (b \rightarrow \text{Meas} (a, b)) \rightarrow b \rightarrow \text{Meas} (\text{Stream} a)
\]
\[
\text{iidM} :: \text{Meas} \text{RealNum} \rightarrow \text{Meas} (\text{Stream} \text{RealNum})
\]

For instance,
\[
\text{iidM} m = \text{do} \{ x \leftarrow m ; xs \leftarrow \text{iidM} m ; \text{return} \ (x : xs) \}.
\]

Consider the program
\[
\text{do} \{ \text{xs} \leftarrow \text{iidM} (\text{do} \{ \text{score} 2 ; \text{return} 42 \}) ; \text{return} \ (\text{hd} \ \text{xs}) \} :: \text{Meas} \text{RealNum}
\]
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 \( 2^n \rightarrow \infty \) as \( n \rightarrow \infty \).

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

**Proposition 2.2.** Suppose that there are terms \( \text{unfoldM} \) and \( \text{iidM} \) as in (6) such that we have \( \text{iidM} (\text{return} \ t) = \text{sample} \ (\text{iid} \ (\text{return} \ t)) \). Then the overall score of a terminating program is undecidable.

**Proof.** Pick an arbitrary Turing machine \( M \). We exhibit a program \( h \) such that we have either \( h = \text{score} 0 ; \text{return} 0 \) if \( M \) halts, and \( h = \text{score} 1 ; \text{return} 42 \) otherwise. To this end, write \( H n \) if \( M \) halts after simulating it for \( n \) steps. Suppose we extend the metalanguage with this \( H \) construction.

\[
h :: \text{Meas} \text{Integer}
\]
\[
h = \text{hd} \ <$> (\text{unfold} (\_ \rightarrow \text{do} \{ \text{score} \ (H \ n) \ \text{then} \ 0 \ \text{else} \ 1 \} ; \text{return} \ (42,n+1) \]))) \ 0
\]
where \( (t \ <$> u) = \text{do} \{ x \leftarrow u ; \text{return} \ (t \ x) \} \). Suppose that \( M \) terminates at time \( n \). Then, unrolling \(<$>\) and the recursive definition, and applying simplifications from Fig. 4:

\[
h = \text{do} \{ \text{xs} \leftarrow (\text{unfold} \ldots) \ 0 ; \text{return} \ (\text{hd} \ \text{xs}) \} = \text{do} \{ \text{xs} \leftarrow (\text{unfold} \ldots) \ 1 ; \text{return} 42\}
\]
\[
= \ldots \ = \text{do} \{ \text{xs} \leftarrow (\text{unfold} \ldots) \ n ; \text{return} 42\}
\]
\[
= \text{do} \{ \text{score} \ 0 ; \text{xs} \leftarrow (\text{unfold} \ldots) \ (n+1) ; \text{return} 42\} = \text{do} \{ \text{score} \ 0 ; \text{return} 0\}.
\]

On the other hand, if the machine \( M \) does not halt, then \( h = \text{hd} \ <$> \text{iidM} \ (\text{return} 42) \). By the assumption in the theorem, \( h = \text{hd} \ <$> \text{sample} \ (\text{iid} \ (\text{return} 42)) \), and so by (1) and (4), we have
\[
h = \text{hd} \ <$> \text{do} \{ \text{xs} \leftarrow \text{sample} \ (\text{iid} \ (\text{return} 42)) \} ; \text{return} 42 ; \text{xs} \}
\]
\[
= \text{do} \{ \text{xs} \leftarrow \text{sample} \ (\text{iid} \ (\text{return} 42)) \} ; \text{return} 42 \}
\]
\[
= \text{sample} \ (\text{do} \{ \text{xs} \leftarrow \text{iid} \ (\text{return} 42) \} ; \text{return} 42) = \text{sample} \ (\text{return} 42) \]
\[
= \text{return} 42 = \text{do} \{ \text{score} \ 1 ; \text{return} 42\} \]
\[
\]
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.
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.

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, [3]).

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:

$$x_0 \sim \text{Exponential}(r) \quad (x_{i+1} - x_i) \sim \text{Exponential}(r)$$

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

```haskell
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 [16], 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:

```haskell
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.

---

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

```haskell
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

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

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

```haskell
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:

```haskell
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 [3] 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.

3.3.1 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.
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.

As an example, we consider stick-breaking for a Dirichlet process (e.g. [22]), 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

\[ r_k \sim \text{Beta}(1, \alpha) \quad v_k = r_k \cdot \prod_{i=1}^{k-1} (1 - r_i) \]

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

Stick-breaking is easy to define as a random stream \(\text{stickBreaking} \alpha\) in our metalanguage.

\[
\text{stickBreaking} \alpha \equiv \text{do}\ rs \leftarrow \text{iid}\ \beta(1, \alpha) \\
\text{let}\ vs = \text{zipWith (*) rs} \cdot \text{scanl (*) 1} \cdot \text{map (1-) rs} \\
\text{return vs}
\]

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.

\[
\text{toProb} \equiv \text{do}\ r \leftarrow \text{uniform}; \text{return}\ \text{findIndex (> r) (scanl (+) 0 vs)}
\]

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:

\[
\text{dp} \equiv \text{RealNum} \rightarrow \text{Prob a} \rightarrow \text{Prob (Prob a)} \\
\text{dp} \alpha\ base = \text{do}\ vs \leftarrow \text{stickBreaking} \alpha; \text{xs} \leftarrow \text{iid base}; \text{return}\ \text{do}\ n \leftarrow \text{toProb vs}; \text{return} (xs!!n)}
\]

We can then piece together a Dirichlet Process clustering model.

\[
\text{cluster} \equiv \text{RealNum} \rightarrow (\text{Prob a}) \rightarrow ((b, a) \rightarrow \text{RealNum}) \rightarrow [b] \rightarrow \text{Meas [(b,a)]}}
\]

\[
\text{cluster} \alpha\ base\ likelihood\ dataset = \text{do}\ p \leftarrow \text{sample}\ \text{dp} \alpha\ base \\\n\text{xs} \leftarrow \text{sample}\ \text{iid}\ base \\\n\text{let}\ taggedData = \text{zip dataset xs} \\\n\text{mapM_ (score . likelihood) taggedData} \\\n\text{return}\ taggedData
\]
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 \((1 - \prod_{i=1}^{n} v_i) < \varepsilon\), 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 [50]. 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 features, such as continent, industry, development, conflict.

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

\[
bp :: \text{RealNum} \rightarrow \text{Prob a} \rightarrow \text{Prob (Prob [a])}
\]

that corresponds to a combination of Beta and Bernoulli processes (e.g. [67]). This generalizes the Dirichlet process \(dp :: \text{RealNum} \rightarrow \text{Prob a} \rightarrow \text{Prob (Prob a)}\) (§3.3) by producing a list rather than a single value. See [3] for details.

A note about implementation. Although the two interfaces are similar, some new challenges arise here because the features are unbounded. Informally, in the Dirichlet process we can stop stick-breaking once we have found a cluster; in the Beta process we must keep looking for potential features, and the process does not terminate. These difficulties appear to coincide with continuity issues in the analysis of the Beta process (e.g. [56, Sl. 6], also [1, 54, 55]). In our implementation [3] we avoid these issues by making use of some amount of hidden state. We set up an Indian Buffet process [28], 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 [67]. The state is safely encapsulated (see e.g. [?]), 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 \texttt{iid} construction (§2.3). The type of streams is often isomorphic to the function type \texttt{Nat \to 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 \texttt{Prob} in exploring function types.

We have seen random functions \texttt{linear} and \texttt{piecewiseLinear} :\texttt{Prob (RealNum \to 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 [3].

4.1 Gaussian process regression

A Gaussian process (e.g. [22, Ch. 11]) is a random function $f$ with the property that for a finite input sequence $(x_1 \ldots x_n)$ the output distribution $(f(x_1) \ldots f(x_n))$ 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 $f \sim \text{GP}(m, k)$, then for all $x_0 \in \mathbb{R}$, $f(x_0) \sim \text{Normal}(m(x_0), \sqrt{k(x_0, x_0)})$ and $f'(x_0 \mapsto y) \sim \text{GP}(m|_{x_0 \mapsto y}, k|_{x_0 \mapsto y})$

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

\begin{verbatim}
  gp :: (RealNum -> RealNum) -> (RealNum -> RealNum -> RealNum) -> Prob (RealNum -> RealNum)
\end{verbatim}

and consider the following equation at type \texttt{Prob (RealNum \to RealNum)}, for any $x_0$:

\begin{verbatim}
  gp m k = do y <- normal (m x0) (sqrt (k x0 x0))
              f <- gp (m' x0 y) (k' x0 y)
              return (\x -> if x==x0 then y else f x)
\end{verbatim}

where $m' \cdot x_0 \cdot y$ and $k' \cdot x_0 \cdot y$ are the conditional mean and covariance functions. To lazily evaluate a Gaussian process we can use (7) to unroll it as necessary, and then use the affine property of the probability monad (3) to remove any reference to \texttt{gp} (c.f. Theorem 2.1).

Once \texttt{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:

\begin{verbatim}
  gpWithLinear :: Prob (RealNum -> RealNum)
  gpWithLinear = do { f <- linear ; return $ gp f (rbf 1 1) }
\end{verbatim}

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

\textit{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 \texttt{gp} in LazyPPL [3] by first making an infinite sequence of standard normals, using \texttt{iid (normal @ 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 \texttt{Prob (2,3)} remain satisfied.

\begin{verbatim}
^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(x_0, u)k(x_0, v)}{k(x_0, x_0)}.
\end{verbatim}
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 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’ [15], 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 [43]. In functional probabilistic programming, memoization becomes a powerful method for building infinite-dimensional probability measures (e.g. [24, 54, 75]). 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 \(F: A \rightarrow B\) with

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

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

\[
\text{memoize} :: (a \rightarrow \text{Prob} b) \rightarrow \text{Prob} (a \rightarrow b)
\]

The idea is that \text{memoize} \(p::\text{Prob} (a \rightarrow b)\) 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 \text{Prob} (a \rightarrow b):

\[
\text{memoize} p = \text{do} \{ y \leftarrow p x_0; f \leftarrow \text{memoize} p; \text{return} \ (\lambda x \rightarrow \text{if } x==x_0 \text{ then } y \text{ else } f x)\}(8)
\]

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 \text{memoize}.

4.2.1 Memoization on countable types. By regarding \(\text{Nat} \rightarrow a\) as isomorphic to \(\text{Stream} a\), we can define memoize with domain type \text{Nat} just using basic primitives for lazy monadic streams, according to the following equation:

\[
\text{memoize} :: (\text{Nat} \rightarrow \text{Prob} b) \rightarrow \text{Prob} (\text{Nat} \rightarrow b)
\]

\[
\text{memoize} f = \text{do} \{ \text{ys} \leftarrow \text{mapM} f [0..] ; \text{return} \ (\lambda x \rightarrow \text{ys} !! x)\}
\]

(In practice [3], we define \text{memoize} more efficiently using tries; c.f. [30].) We can see, in particular, that \text{memoize} \((\_ \rightarrow \text{p})\) corresponds to \(\text{iid} \text{p}\). With this in mind, following Proposition 2.2, we
cannot hope to have \texttt{memoizeMeas} :: \((a \to \text{Meas } b) \to \text{Meas } (a \to b)\), 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,

\[
\text{dp } \alpha \text{ uniform} :: \text{Prob } (\text{Prob } \text{RealNum})
\]

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,

\[
\forall r. f_1(r) \sim \text{Normal}(0, 3) \text{ and } f_2(r) \sim \text{Normal}(0, 3).
\]

This can be implemented in our metalanguage by using memoization:

\[
p \leftarrow \text{dp } \alpha \text{ uniform}; \text{xpos} \leftarrow \text{memoize } (\_ \to \text{normal } 0 3); \text{ ypos} \leftarrow \text{memoize } (\_ \to \text{normal } 0 3)
\]

If we return \(\{r \leftarrow p ; \text{return } (\text{xpos } r, \text{ ypos } r)\}\), then this whole program is equivalent to using a different base distribution, \(\text{dp } \alpha (\text{mvnormal } (0,0) (3,3))\). However, the memoized form is more compositional: we can later assign other attributes to clusters in another part of the model (see [3]).

5 INTERPRETATION OF THE PROBABILITY MONAD USING INFINITE TREES

We now provide a model of the metalanguage from Section 2, with two monads: \text{Prob} and \text{Meas}. Our model is based on quasi-Borel spaces [29]. 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, [3]).

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. [40]), 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. \(\text{ob}(\mathcal{V}) = \text{ob}(C)\)).

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 \text{Meas}. The composition in \(C\) corresponds to the sequencing of the metalanguage. First, a context \(\Gamma = (x_1 :: a_1, \ldots, x_n :: a_n)\) is interpreted as the monoidal product \(a_1 \otimes \cdots \otimes a_n\) in \(C\). Then a term \(\Gamma \vdash t :: \text{Meas } a\) can be interpreted as a Kleisli morphism, i.e. a morphism \(\Gamma \to a\) in \(C\). If we also have \(x :: a, \Delta \vdash u :: \text{Meas } b\), then the sequencing \(\Gamma, \Delta \vdash \text{do } \{x \leftarrow t ; u\} :: \text{Meas } b\) is the following composite morphism:

\[
\Gamma, \Delta \xrightarrow{t \otimes \Delta} a, \Delta \to \text{Meas } b
\]
With this in mind, the commutativity law (2) amounts to the interchange law of monoidal categories, \((a' \otimes g) \circ (f \otimes b) = (f \otimes b') \circ (a \otimes g)\) for \(f: a \to a'\) and \(g: b \to b'\).

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

5.2 Key intuition: randomized functions and splitting

Let \(\Omega\) be a set of random seeds. A randomized function between sets \(X\) and \(Y\) is a function \(f: X \times \Omega \to Y\), that depends on the random seed. Suppose that we have a method for splitting random seeds, \(\gamma: \Omega \to \Omega \times \Omega\) (e.g. [64]). Then we can compose randomized functions

\[
f: X \times \Omega \to Y \quad g: Y \times \Omega \to Z \quad (g \circ f): X \times \Omega \to Z
\]

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 \(f: X \times \Omega \to Y\) as a function \(X \times Y^\Omega\). Once we equate certain functions, \(Y^\Omega\) 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 \((\Omega, \gamma)\) 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 \((X, \Sigma_X)\) is a set \(X\) together with a set \(\Sigma_X\) of ‘measurable subsets’ of \(X\), which must be a \(\sigma\)-algebra, i.e. closed under countable unions and complements. A measure on a space \((X, \Sigma_X)\) is a function \(\mu: \Sigma_X \to [0, \infty]\) that is \(\sigma\)-additive \((\mu(\bigcup_{i=1}^\infty U_i) = \sum_{i=1}^\infty \mu(U_i))\); it is a probability measure if it is normalized, i.e. if \(\mu(X) = 1\). A function \(f: (X, \Sigma_X) \to (Y, \Sigma_Y)\) 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 \(\sigma\)-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 \(\mu\) on \((X, \Sigma_X)\), we can find the expected value of any measurable function \(f: (X, \Sigma) \to (\mathbb{R}, \Sigma_{\mathbb{R}})\), notated \(\int f(x) \mu(dx) \in [0, \infty]\), the Lebesgue integral of \(f\) with respect to \(\mu\). 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. [33]).** (1) A standard Borel space is a measurable space \((X, \Sigma_X)\) that is either (a) countable, with \(\Sigma_X\) 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 \(\Omega\) be a fixed uncountable standard Borel space (traditionally \(\Omega = \mathbb{R}\), but see Section 5.4.1).

**Definition 5.4 ([29]).** 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 \(f: \Omega \to X\) is measurable then \((\alpha \circ f) \in M_X\);
proposition 5.5 ([29]). Quasi-borel spaces and morphisms form a category $\text{Qbs}$ that is cartesian closed. Standard Borel spaces $(X, \Sigma_X)$ fully embed in $\text{Qbs}$, 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 $(\Omega, \Sigma_\Omega)$ with a probability measure $\mu$ on it;
- a measure-preserving function $\gamma : (\Omega, \mu) \to (\Omega \times \Omega, \mu \otimes \mu)$.

i.e. for all $f : \Omega \times \Omega \to \mathbb{R}$, $\int f(\gamma(\omega)) \, \mu(\mathrm{d}\omega) = \int \int f(\omega_1, \omega_2) \, \mu(\mathrm{d}\omega_2) \, \mu(\mathrm{d}\omega_1)$;

- a chosen uniformly distributed random variable $\nu : \Omega \to [0, 1]$.

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

$\gamma(\omega) = (\lambda(i_1, \ldots, i_n), \omega(0, i_1, \ldots, i_n), \lambda(i_1, \ldots, i_n), \omega(i_1 + 1, \ldots, i_n))$

In fact, this $\gamma$ 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 $\gamma$ splits the tree as indicated by the dotted line:

```
\omega(\cdot) \quad \omega(1) \quad \omega(2) \quad \omega(3) \quad \cdots
\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \cdots
\omega(0,0) \quad \omega(0,1) \quad \omega(0,2) \quad \omega(1,0) \quad \omega(1,1) \quad \omega(1,2) \quad \cdots
```

Our probability measure $\mu$ on this choice of $\Omega$ 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 $\nu : \Omega \to [0, 1]$.

5.4.2 Probability kernels.

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

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

(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 $\nu : \Omega \to [0, 1]$.

• 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:

$$A \times X \times \Omega \xrightarrow{A \times X \times Y} A \times X \times \Omega \times \Omega \xrightarrow{\pi} A \times \Omega \times X \times \Omega \xrightarrow{f \times g} B \times Y$$

**Proposition 5.7.** Probability kernels modulo equivalence (9) form a monoidal category $\text{ProbKer}$: 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 $(X \times \Omega \xrightarrow{\text{fst}} X \xrightarrow{f} Y)$; this induces an identity-on-objects functor $\text{Qbs} \to \text{ProbKer}$.

**Proposition 5.8 (see [29]).** The inclusion functor $\text{Qbs} \to \text{ProbKer}$ 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 $(\sqrt{2\pi})e^{|x|^2}$ defines the Lebesgue measure on $\mathbb{R}$, which assigns to each open interval its length (see also [63]).

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 $g : Y \to \mathbb{R}$,

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

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 $\text{Qbs} \to \text{MeasKer}$.
We also have the concrete distributions used in the examples in Section 3. All distributions on \( \mathbb{R} \) are of the form \( \text{Prob} \mathbb{R} \) with \( \mathbb{R} \) being the standard probability space, which is well-known to be problematic. By contrast, an infinite-dimensional uniform probability distribution is straightforward, by Kolmogorov extension, and is very useful.

The \( \text{Prob} \mathbb{R} \) and \( \text{Meas} \) monads are induced by the category of probability kernels, and is affine and commutative, by Proposition 5.8.

The morphism \( \text{sample} : \text{Prob} \rightarrow \text{Meas} \) is induced by the inclusion of the category of probability kernels into the category of measure kernels, taking constant weights \( \ell = 1 \).

The morphism \( \text{score} : \text{Meas} \rightarrow \text{Meas} \) is induced by putting \( \ell : \mathbb{R} \times \Omega \rightarrow [0, \infty] \) 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 \( \text{idMeas} : \text{Meas} \rightarrow \text{Meas} \) is defined by the Kolmogorov extension theorem.

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

Categories of measure kernels. The probability measures on a measurable space \( X \) form a commutative monoid on the category of measurable spaces and measure-preserving maps, called the Giry monad [23]. Moreover, the s-finite measure kernels between measurable spaces form a monoidal category [62]. This more traditional measure-theoretic foundation forms a good intuition for our metalanguage (§2) (see also [39]), 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 [4]; 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.

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. [61]; see also [41, 60]). Let us briefly define a para-randomized function between sets \( X \) and \( Y \) to be a pair \((\Omega, f)\), where \( \Omega \) is a standard probability space and \( f : X \times \Omega \rightarrow Y \) 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

\[
    f : X \times \Omega_1 \rightarrow Y \quad g : Y \times \Omega_2 \rightarrow Z \quad (g \circ f) : X \times (\Omega_1 \times \Omega_2) \rightarrow Z
\]

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 [72]. We omit the details, which are largely orthogonal.

A different approach would be to use the recent work of [26] to build an entirely domain theoretic model. There are also other potential ways to interpret laziness in probabilistic models of linear logic [17]. 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, do we 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 [3]. For the probability space, we put $\Omega = \text{Tree}$ and $\gamma = \text{splitTree}$:

```hs
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 $\text{Tree} \to a$.

```hs
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 [12, §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.

```hs
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 [3]. A simple reference method is a likelihood-weighted importance sampler, following e.g. [70, §4.1], which is just 20 lines of Haskell. This has type $\text{lwis} :: \text{Int} \to \text{Meas} a \to \text{IO} [a]$. Running $\text{lwis} n m$ 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 $\text{Meas} X$ induces a pair of functions $\mathbb{R} \overset{\ell}{\leftarrow} \Omega \overset{f}{\to} X$, where $\Omega$ is regarded with a basic probability measure $p$, and $\ell$ is
measurable. Here \( \ell \) is regarded as a density for an unnormalized distribution on \( \Omega \), 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 \( \mu \) on \( \Omega \);
- The unnormalized measure \( \mu_\ell \) on \( \Omega \), induced by regarding \( \ell \) as a density. Formally, \( \mu_\ell(U) = \int_\Omega [\omega \in U] \cdot \ell(\omega) \mu(d\omega) \). This could be written in the metalanguage as

  \[
  \text{do } \{ \omega \leftarrow \text{sample } \mu; \text{score } (\ell \omega); \text{return } \omega \} :: \text{Meas } \Omega.
  \]

(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 \textit{normalized} form of the measure \( \mu_\ell = \frac{\mu_\ell}{\mu(\Omega)} \), which is a probability measure, assuming \( \mu(\Omega) \in (0, \infty) \).
- The pushforward probability measure on \( X \), \( f^*(\frac{\mu_\ell}{\mu(\Omega)}) \). This could be written as

  \[
  \text{do } \{ \omega \leftarrow \text{sample } \mu; \text{score } (\ell \omega); \text{score } (1/(\mu(\Omega)); \text{return } (f \omega) \} :: \text{Meas } X.
  \]

The challenge is that the normalizing constant \( \mu(\Omega) \) is typically difficult to calculate. The Markov-Chain Monte Carlo simulation algorithms provide a sampling procedure for \( \frac{\mu_\ell}{\mu(\Omega)} \) on \( \Omega \), without explicitly calculating \( \mu(\Omega) \). They are best described as algorithms over \( \Omega \), 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 \( k : \Omega \times \Sigma_\Omega \rightarrow [0, 1] \) such that each \( k(\omega, -) \) is a probability measure and each \( k(-, U) \) is measurable. We follow the analysis of proposal kernels from [21, 27].

The proposal kernel \( k \) does not directly capture the probability measure \( \frac{\mu_\ell}{\mu(\Omega)} \). 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 \( \mu_\ell \). This gives two measures \( m, m_{\text{rev}} \) on \( \Omega \times \Omega \):

\[
m(U) = \int_\Omega \int_\Omega [\omega_1, \omega_2] \in U \cdot \ell(\omega_1) k(\omega_1, d\omega_2) \mu(d\omega_1)
\]

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

These can be described in programming terms as

\[
m = \text{do } \{ \omega_1 \leftarrow \text{sample } \mu; \omega_2 \leftarrow \text{sample } (k \omega_1); \text{score } (\ell \omega_1); \text{return } (\omega_1, \omega_2) \}
\]

\[
m_{\text{rev}} = \text{do } \{ \omega_1 \leftarrow \text{sample } \mu; \omega_2 \leftarrow \text{sample } (k \omega_1); \text{score } (\ell \omega_1); \text{return } (\omega_2, \omega_1) \}
\]

**Definition 6.1 ([27]).** We say that a kernel \( k \) is \textit{Green} with respect to \( \ell \) and \( \mu \) if \( m_{\text{rev}} \) is absolutely continuous with respect to \( m \). This means that there exists a ‘ratio’ \( r : \Omega \times \Omega \rightarrow \mathbb{R} \) (the ‘Radon-Nikodym derivative’) such that

\[
\int [\omega_1, \omega_2] \in U \cdot r(\omega_1, \omega_2) \cdot \ell(\omega_1) k(\omega_1, d\omega_2) \mu(d\omega_1) = \int [\omega_2, \omega_1] \in U \cdot \ell(\omega_1) k(\omega_1, d\omega_2) \mu(d\omega_1)
\]
or in programming terms
\[
\begin{align*}
\text{do } \{ \omega_1 & \leftarrow \text{sample } \mu \; ; \; \omega_2 \leftarrow \text{sample } (k \omega_1) \; ; \; \text{score } (\ell \omega_1) \; ; \; \text{score } (r \omega_1 \omega_2) \; ; \; \text{return } (\omega_1, \omega_2) \} \\
= & \text{do } \{ \omega_1 \leftarrow \text{sample } \mu \; ; \; \omega_2 \leftarrow \text{sample } (k \omega_1) \; ; \; \text{score } (\ell \omega_1) \; ; \; \text{return } (\omega_2, \omega_1) \}
\end{align*}
\]

6.2 A new proposal kernel for lazy rose trees

Recall our choice of \( \Omega \) is rose trees: infinitely deep and infinitely wide trees labelled from \([0, 1] \), with the basic probability measure \( \mu \) 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.

```haskell
mutateTree :: RealNum \rightarrow Tree \rightarrow Prob Tree
mutateTree p (Tree a ts) = do b \leftarrow bernoulli p
  a' \leftarrow uniform
  ts' \leftarrow 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 \( k : \Omega \times \Sigma \Omega \rightarrow [0, 1] \) given by \((\text{mutateTree } p)\) is Green, and the ratio is

\[
r(\omega_1, \omega_2) = \frac{f(\omega_2)}{f(\omega_1)}.\]

**Proof notes.** Notice that \( k \) is reversible with respect to \( \mu \) in that

\[
\text{do } \{ \omega_1 \leftarrow \mu \; ; \; \omega_2 \leftarrow k \omega_1 \; ; \; \text{return } (\omega_1, \omega_2) \} = \text{do } \{ \omega_1 \leftarrow \mu \; ; \; \omega_2 \leftarrow k \omega_1 \; ; \; \text{return } (\omega_2, \omega_1) \}
\]

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

\[
\begin{align*}
\text{do } \{ \omega_1 & \leftarrow \text{sample } \mu \; ; \; \omega_2 \leftarrow \text{sample } (k \omega_1) \; ; \; \text{score } (\ell \omega_1) \; ; \; \text{score } (r \omega_1 \omega_2) \; ; \; \text{return } (\omega_1, \omega_2) \} \\
= & \text{do } \{ \omega_1 \leftarrow \text{sample } \mu \; ; \; \omega_2 \leftarrow \text{sample } (k \omega_1) \; ; \; \text{score } (\ell \omega_1) \; ; \; \text{return } (\omega_1, \omega_2) \} \\
= & \text{do } \{ \omega_1 \leftarrow \text{sample } \mu \; ; \; \omega_2 \leftarrow \text{sample } (k \omega_1) \; ; \; \text{score } (\ell \omega_1) \; ; \; \text{return } (\omega_2, \omega_1) \}
\end{align*}
\]

as required, where the second step uses the reversibility of \( k \) with respect to \( \mu \). \( \square \)

**Technical note.** Our \( k \) is reversible in the given sense, and this appears to be a 'Metropolis ratio'. But because our space \( \Omega \) 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 \( k : \Omega \times \Sigma \Omega \rightarrow [0, 1] \) be a Green Markov kernel with ratio \( r : \Omega \times \Omega \rightarrow [0, 1] \). The Metropolis-Hastings-Green kernel \( k_{\text{MHG}} : \Omega \times \Sigma \Omega \rightarrow [0, 1] \) is now given by proposing a new \( \omega_2 \) via \( k(\omega_1, \cdot) \), and accepting or rejecting the proposal according to \( \min(1, r(\omega_1, \omega_2)) \). Either way, we produce something, either the new \( \omega_2 \) or the old \( \omega_1 \).

```haskell
k_{\text{MHG}} :: (\Omega \rightarrow Prob \Omega)
k_{\text{MHG}} \omega_1 =
  \text{do } \{ \omega_2 \leftarrow k \omega_1 \; ; \; b \leftarrow \text{bernoulli } min 1 (r \omega_1 \omega_2) \; ; \; \text{if } b \text{ then return } \omega_2 \text{ else return } \omega_1 \}
```
We can then construct a Markov chain with transitions given by \( k_{\text{MHG}} \). The key result (e.g. [21, 27]) 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 \( \nu \) on \( \Omega \) is a stationary distribution for a kernel \( k : \Omega \times \Sigma_\Omega \rightarrow [0, 1] \) if
\[
\int_{\Omega} k(\omega, U) \, \nu(d\omega) = \nu(U).
\]
We say that \( k \) is irreducible with respect to a probability measure \( \xi \) 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 \( k_{\text{MHG}} \) has a stationary distribution, which is the normalized probability measure \( \frac{\mu}{\mu_T(\Omega)} \) on \( \Omega \). If \( k_{\text{MHG}} \) is irreducible with respect to \( \frac{\mu}{\mu_T(\Omega)} \) 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 \texttt{mutateTree} kernel (§6.2) with \( p = 1 \), \( k_{\text{MHG}} \) is irreducible for \( \frac{\mu}{\mu_T(\Omega)} \).

**Proof note.** Here \( n = 1 \) suffices. \( \square \)

We recall that correctness of a similar ‘all-sites’ Metropolis-Hastings scheme for probabilistic programming was proved in [7], albeit for a non-lazy language.

There remains a concern that \( k_{\text{MHG}} \) 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 \( \omega \), 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, \texttt{(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 \( \texttt{(mutateTree p)} \) (§6.2). Each step of the algorithm provides a sample from the measure \( \frac{\mu}{\mu_T(\Omega)} \) on \( \Omega \), and we can push-forward this sample along \( f : \Omega \rightarrow X \) 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 (\texttt{mutateTree p}) with (\texttt{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. [74] (§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

\[ c \in (0, 1) \]

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. [74] (§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 [21, 27]. Let \( k_i : \Omega \times \sum_\Omega \rightarrow [0, 1] \) be a countable family of Markov kernels (§6.1), and let \( c : \Omega \times \sum_\Omega \rightarrow [0, 1] \) be a parameterized probability distribution function over \( \sum_\Omega \), i.e. for all \( \omega \in \Omega \),

\[ \sum_{i=1}^\infty c(\omega, i) = 1. \]

Let \( k \) be the mixed kernel

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

Suppose that each \( k_i \) is a Green kernel with respect to \( \mu \), with ratio \( r_i : \Omega \times \Omega \rightarrow [0, \infty] \) (Def. 6.1).

Suppose that we can always detect which kernel was used, i.e. there is a function \( e : \Omega \times \sum_\Omega \rightarrow \sum_\Omega \) such that \( \int [e(\omega, \omega') = i] k_i(\omega, d\omega') \mu(d\omega) = 1. \)

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

\[ r(\omega, \omega') = r_i(\omega, \omega') \cdot \frac{c(\omega', i)}{c(\omega, i)} \quad \text{where } i = e(\omega, \omega'). \]

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, use the same \( n \) for \( (r \cdot k + (1 - r) \cdot k') \) as for \( k \).

In particular, completing the discussion from Section 6.3, \( \texttt{mutateTree p} \) might not induce an irreducible kernel, but if an irreducible kernel is desired then we can instead start from the mixed kernel \( (r \cdot \texttt{mutateTree 1} + (1 - r) \cdot \texttt{mutateTree p}) \), 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 ([74],[70, §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 \( \Omega \) 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 \( \omega \) and \( \omega' \) differ by only one node, then let \( c(\omega, \omega') \) returns the path to this node.
- For any given tree \( \omega \), we define \( c(\omega, i) \) as follows. First, if \( \ell \) and \( f \) are defined by programs that terminate, then we calculate the necessarily finite set of nodes \( S_\omega = \{i_1 \ldots i_n\} \) 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 \( \frac{\ell(\omega')|S_\omega|}{\ell(\omega)|S_{\omega'}|} \).

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 \( S_\omega \).

The \texttt{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 \[70\], 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.

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 \[24\] 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 \( \text{do } \{ x \leftarrow t; \ y \leftarrow u; \ z \leftarrow v; \ldots \} \) and expecting lazy evaluation, one can write (roughly) \( f \leftarrow \text{memoize}(\lambda i \to \text{case } i \text{ of } \{ 1 \to t; \ 2 \to u; \ 3 \to v \}) \); \ldots 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 useable 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 \[24, 54\] and exchangeable primitives \[76\]. In summary, from a bird’s eye view, our metalanguage (§2) and
implementation [3] form a variation of Church with more idiomatic laziness, a type system and a
semantics.

Languages such as Anglican [69], WebPPL [25], BayesDB [57] and Turing [6] follow within the
tradition of Church, exploring ideas from non-parametric statistics further. The Birch language [47]
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 [45, 46].

Beyond these examples, laziness has been explored in various aspects of probabilistic program-
mapping, dating back at least as far as the pioneering work by Koller et al. [38], and more recently
in the work on lazy factored inference in Figaro [52], and efficient implementation in delimited
continuations through Hansei ([34], 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 [48, 49, 73] provides
a DSL with impressive symbolic inference methods. Stochaskell [53] 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 [59], which is a monad-based implementa-
tion of a variety of inference combinators, also inspired by the formalism of quasi-Borel spaces [60].
MonadBayes is not fully lazy: the Metropolis-Hastings simulation is based on the state monad and
does not support laziness. One starting point for this paper was the problem of allowing laziness in
MonadBayes, leading to the compositional expression of the various examples in Sections 3 and 4.
For simplicity of exposition, we have focused on Metropolis-Hastings inference, but in practice it
would be appropriate to adapt some of the other inference combinators of MonadBayes to the lazy
setting too.

Going beyond the inference combinators of MonadBayes, quasi-Borel spaces are also a foundation
for a new dependent type system based on ‘trace types’ [41] (prototyped in Haskell). This provides
a well-typed account of the ‘programmable inference’ that makes recent languages such as Gen [14]
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. [2, 58]) and functional languages for
probabilistic network verification ([71], following [18]).

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 [9], a python library inspired by the effectus theory foundation for probability [11].
The second is a python/F# library for exact conditioning over Gaussian-based models [65], inspired
by categorical constructions over Markov categories [19, 66]. 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.

REFERENCES


