LazyPPL: Monads and laziness for Bayesian modelling

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

We introduce LazyPPL (Lazy Probabilistic Programming Library), a Haskell library for Bayesian modelling that supports laziness. This is intended to experiment with recent developments in 'synthetic measure theory', such as quasi-Borel spaces and Markov categories. The idea is that types are spaces, and programs are measures on those spaces. As we demonstrate through numerous examples, laziness is a very natural idiom for statistical modelling. To perform Bayesian inference with these examples reasonably efficiently, we introduce two new inference methods that are based on the Metropolis-Hastings-Green family of algorithms, but adapted to laziness.

1 INTRODUCTION

In this paper we analyze the crucial role that laziness can play in expressive Bayesian modelling. To this end we provide two new Metropolis-Hastings algorithms (§1.4, §5, §7) that work in a lazy setting, and demonstrate their expressive power through a wealth of examples (§3, §6; source code is provided in supplementary material). As we now explain, our development is motivated by new compositional methods in measure theory, such as quasi-Borel spaces and Markov categories (§4), and we bring these together with recent practical advances on probabilistic programming (see §8 for a summary).

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. [20, §1.1], [21, §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. [20, 71]). 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 [41], Church [24], Stan [8] and others) can often be regarded as programming languages describing unnormalized measures, that are endowed with efficient Monte Carlo samplers.

On the theoretical side, researchers have recently proposed synthetic probability theory [10, 16] and synthetic measure theory [29, 38, 61], with the aim of developing compositional structures in measure theory. There are various aims in that work, some axiomatic, and some seeking to sidestep cumbersome issues with measure theory, such as the absence of function spaces and of a strong monad of measures (see §4). In this way, probabilistic programming can be viewed as a *practical synthetic measure theory*, with compositionality built in, and where types are spaces, and programs

Authors' address: Swaraj Dash, swaraj.dash@cs.ox.ac.uk; Younesse Kaddar, younesse.kaddar@cs.ox.ac.uk; Hugo Paquet, hugo.paquet@cs.ox.ac.uk; Sam Staton, sam.staton@cs.ox.ac.uk,
Department of Computer Science, University of Oxford, UK.

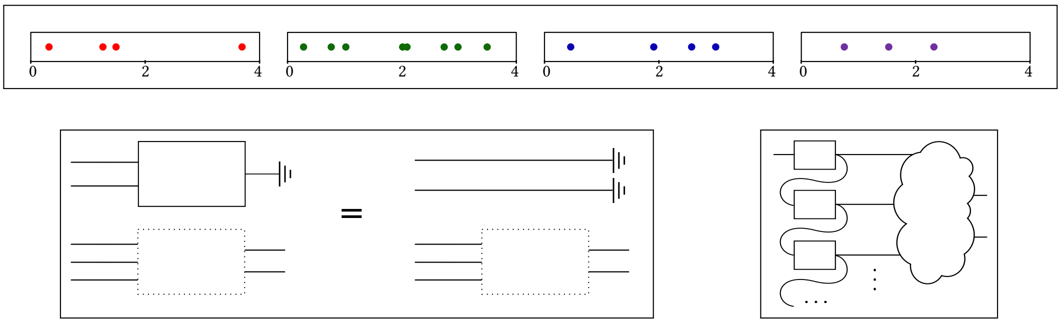


Fig. 1. (a) Four samples from a 1D Poisson point process with rate 1, with viewport restricted to $[0, 4]$. (b) The law for affine monoidal categories in string diagram form. (c) Visualizing dataflow in a lazy infinite process.

are suitably good measures on the spaces. By exploring fully expressive probabilistic programming languages, we are exploring the abstract and unusual 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 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 an implementation is given in §3.1.2.

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 3.1 we use it as part of a regression problem) and then it is less appropriate to truncate it to an arbitrary viewport in advance. 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 give other examples of non-parametric processes in Sections 3.2 (Clustering and Chinese Restaurant), 6.4 (Wiener), 6.5 (Indian Buffet), and 6.6 (Mondrian Process).

1.3 Theoretical aspects: affine monoidal structure and synthetic spaces

To illustrate the theoretical side, we recall that synthetic foundations of probability theory are often based on affine monoidal categories or affine monads (e.g. [10, 13, 16, 17, 32, 62]). Here *affine* means that the monoidal unit is terminal, shown diagrammatically in Figure 1 (b). In integral notation, this amounts to the equation $\int 1\,\mu(\mathrm{d}x) = 1$ for probability measures μ. 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 cloud represents whatever happens next, and intuitively, those morphisms that are not used in what follows need never be inspected, and in that case the process can likely be truncated. Thus, affine monoidal categories are related to laziness.

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. Moreover, we see new spaces, such as an abstract space of the Tables of the Chinese Restaurant Process (§3.2.3), which encapsulates crucial aspects of exchangeability from non-parametrics (see also [65, 66]).

1.4 New Metropolis-Hastings-based inference algorithms

To experiment with these examples involving laziness, we introduce two new inference algorithms that build on earlier inference methods for probabilistic programming (e.g. [76]). 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, we cannot necessarily ask for the dimension of the state space without triggering suspended execution paths. We provide two new correct instantiations of the Metropolis-Hastings-Green algorithm:

- Our first algorithm (§5) is purely lazy. It operates lazily over the infinite-dimensional state space, mutating different parts at random. In fact, it is so lazy that it only does anything when a plotting or printing routine is invoked, and even then, it will typically only perform the inference needed for the viewport.
- Our second algorithm (§7) uses Haskell internals (`ghc-heap`) to identify which dimensions are actually being used in a given run of the program, and changes exactly one of these dimensions, chosen at random. This is inspired by [76], but adapted to the lazy setting.

We can show that these algorithms are correct (via Theorems 5.2, 7.1, 5.3). The structure of our implementation is based on ideas from quasi-Borel spaces [29]. In particular, we work with a fixed basic probability space Ω, here instantiated to rose trees (§5.2). We base our implementation on two monads from quasi-Borel spaces: `Prob`, for probability measures, which is affine hence lazy, and `Meas`, for unnormalized measures, which are not affine. While both algorithms are useful for prototyping, no general purpose inference algorithm should be expected to perform optimally on all models, and we discuss issues in Sections 5.4 and 7.2.3. That said, our simple general purpose algorithms are actually very useful for the numerous examples we consider in this paper to illustrate laziness in probabilistic programming (§3, §6).

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

2 MONADS 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. We encapsulate this in programming terms by using two monads, describing normalized and unnormalized measures.

- A probability monad `Prob` (often in green font) so that (`Prob a`) intuitively contains probability measures on `a`. This supports an operation

  ```
  uniform :: Prob RealNum
  ```

  which is thought of as providing a fresh sample uniformly distributed between 0 and 1. (We use `RealNum` as a synonym for `Double`, to emphasise the view of types as spaces.) The 'noise outsourcing lemma' states that this uniform draw can be used as a seed for any distribution. For example, we can draw from a standard normal distribution by

  ```
  do { x ← uniform ; return (probit x) }
  ```

  where the probit function is shown in Figure 2 (a). (LazyPPL comes included with implementations of many useful probability measures on `RealNum`, such as `normal`, `beta`, `exponential`, and also measures on other types such as the `bernoulli` distributions on `Bool`.)

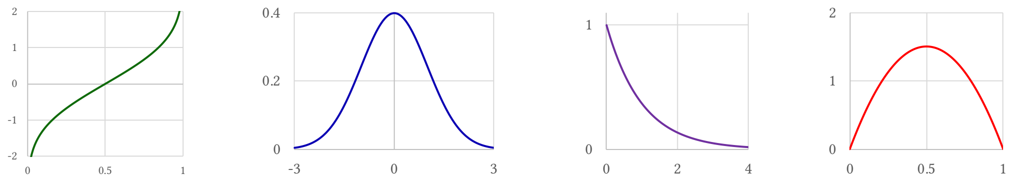


Fig. 2. From left to right: (a) The probit function (inverse cumulative distribution function for the standard normal distribution); then the densities of the (b) standard normal distribution; (c) exponential distribution with rate 1; (d) beta (2,2) distribution.

- A measures monad `Meas` (often in red font), so that (`Meas a`) intuitively contains unnormalized measures on `a`. This supports two operations:
  - `sample :: Prob a → Meas a`, which draws a sample from a probability distribution;
  - `score :: RealNum → Meas ()`, which weights this run of the program by a given weight.

  The `score` operation is often used with a probability density, to incorporate the likelihood in a Bayesian scenario. One can use all kinds of distributions for observations, using their densities (e.g. Fig. 2 (b,c,d)). For example, to incorporate a Bayesian observation of data point `x` from a normal distribution with mean `mu` and standard deviation `sigma`, we write

  ```
  score (normalPdf mu sigma x)
  ```

We provide worked examples using these operations in Section 3. For now, we discuss these operations abstractly.

2.1 Desiderata for probability and measures monads

A strong monad is a structure with a bind and return

```
(>>=) :: m a → (a → m b) → m b        return :: a → ma.
```

This allows us to sequence operations, and Haskell provides the do notation for this. These should satisfy associativity and identity laws (e.g. [43]). Ideally, the measures monad should also satisfy commutativity (e.g. [36]): for `mx :: m a` and `my :: m a`,

```
mx >>= \x → my >>= \y → (x,y)    =    my >>= \y → mx >>= \x → (x,y)        (1)
```

The probability monad should satisfy commutativity and also affinity (e.g. [31, 37]): if `mx :: m a` then:

```
mx >>= \x → return ()    =    return ()        (2)
```

In programming terms, affinity is a substantial requirement: it means that in evaluating `mx >>= f` we need not strictly evaluate `mx`, rather we ought to treat it lazily. Thus a crucial fundamental axiom for probability theory corresponds to the concept of laziness. (See also Fig. 1 (b,c).)

Aside: Understanding in terms of Kleisli categories. If we start from a cartesian closed category C, a strong monad induces another category, a Kleisli category, with a functor $J\colon C \to \mathsf{Kl}(\texttt{m})$. When `m` is commutative, the Kleisli category has a monoidal structure and J preserves it. When `m` is moreover affine, the monoidal structure has a terminal unit. We note that monoidal categories with a terminal unit have been used as a synthetic axiomatization of probability theory (e.g. [10, 16, 18]). In this way, programming with a commutative affine monad is an equivalent way of reasoning about synthetic probability theory (see also [69]).

2.2 Candidate monads

In Section 4 we discuss our probability monads in detail. For now, we remark on some difficulties, and suggest some well known monads that form reasonable candidates.

2.2.1 Candidate probability monads. The following conventional monads might be regarded as probability monads:

- The state monad (`s → (a, s)`) with state `s = Stream RealNum`. The idea is that this is to be run with a stream of random seeds. Then `uniform` takes the head of the stream and puts the tail of the stream back in to the state.
- The continuations monad (`(a → r) → r`) with return type `r = RealNum`. Then `uniform` finds the expected value of its argument somehow, for example by calling it with many values and taking the mean average.

These monads do not satisfy commutativity or affinity. However, if we move to a semantic setting, we can restrict the continuations monad to the integration operators (§2.2.3) via the Giry monad.

2.2.2 Candidate measures monads. The following conventional monads might be regarded as measures monads:

- If `m` is a probability monad, then the writer monad transformer `WriterT RealNum m a` is a measure monad. Here we accumulate real numbers by multiplication:

  ```
  ((r,mx) >>= f)  =  mx >>= \x → (r * (fst (f x)), snd (f x))
  ```

  This writer monad is commutative if `m` is, because multiplication of real numbers is commutative.
- The continuations monad (`(a → r) → r`) with return type `r = RealNum` can be regarded as a measures monad. Let `score :: RealNum → Meas ()` be `\r k → r * k ()`.

Although we would like to think of the measures monad `Meas` as a monad of measures, there is no known strong monad of unnormalized measures on the category of measurable spaces. As we explain in Section 4, quasi-Borel spaces form a good semantic setting for monads of measures.

2.2.3 Measure-theoretic monads. Outside of programming, the Giry monad, which can be defined on categories of measurable spaces and metric spaces (e.g. [22, 51]), is commutative and affine. We now recall some rudiments of measure theory, which is a foundation that explains the apparent paradox of sampling from an uncountable space such as $[0,1]$.

Definition 2.1. A measurable space (X,Σ_X) is a set X together with a set Σ_X of 'measurable subsets' of X, which must be a σ-algebra, i.e. closed under countable unions and complements. A measure on a space (X,Σ_X) is a function $\mu\colon \Sigma_X \to [0,\infty]$ that is σ-additive ($\mu(\biguplus_{i=1}^{\infty} U_i) = \sum_{i=1}^{\infty}\mu(U_i)$); it is a probability measure if $\mu(X)=1$. A function $f\colon (X,\Sigma_X)\to(Y,\Sigma_Y)$ is *measurable* if $f^{-1}(U)\in\Sigma_X$ for all $U\in\Sigma_Y$.

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

For any measurable space (X,Σ_X), the set of probability measures PX can be made into a measurable space, with Σ_{PX} the least σ-algebra making $\int f(x)\,[-](\mathrm{d}x) : PX \to [0,\infty]$ measurable for all $f\colon X\to\mathbb{R}$. This is actually a monad, with $\mu{>>=}f$ the measure which is the integration operator taking $g\colon Y \longrightarrow \mathbb{R}$ to the iterated integral

$$\int g(y)\,(\mu \mathbin{>\!\!>\!=} f)(\mathrm{d}y) \quad = \quad \int\int g(y)\, f(x,\mathrm{d}y)\,\mu(\mathrm{d}x).$$

The category of measurable spaces is not cartesian closed, a point we revisit in Section 4. But we can actually still interpret the commutativity (1) and affine laws (2) for the Giry monad, which respectively amount to Fubini's theorem

$$\int\int f(x,y)\,\mu_Y(\mathrm{d}y)\,\mu_X(\mathrm{d}x) \;=\; \int\int f(x,y)\,\mu_X(\mathrm{d}x)\,\mu_Y(\mathrm{d}y)$$

and the unity of a probability measure

$$\int 1\; mx(\mathrm{d}x) = 1$$

However, for arbitrary measures, iterated integration does not work, and Fubini does not hold (see [64] for a programming angle on this). This is arguably because of obscure measures such as the counting measure which have no role in programming or Monte Carlo simulation. It is an open problem to find a commutative monad of measures on the category of measurable spaces that includes the one-point measures and the probability measures. We revisit this in Section 4.3.

2.2.4 Running probabilistic programs. The key idea of probabilistic programming is to describe an unnormalized measure, and then to use an inference procedure to simulate draws from a normalized measure. For the examples in this paper, it is sufficient to have an inference procedure `infer :: Meas a → IO [a]` which takes an unnormalized measure and outputs a stream of samples from it (§4.4.1, §5, §7). As an aside, we note that in general this might take the form of a function `normalize :: Meas a → Prob a`. But the practicalities of `normalize` and 'nested inference' are difficult and not especially widely used (although see e.g. [26, 70, 79]).

3 EXAMPLES TAKING ADVANTAGE OF LAZINESS

We now give examples of probabilistic programs using the interface in Section 2. Where LazyPPL comes into its own is that the models can involve types, regarded as spaces, and structures that have unbounded or infinite dimension. We stress that the models here have been dealt with before in other probabilistic programming languages. Our aim is to emphasise the especially natural way of expressing them in LazyPPL, using Haskell's types and laziness.

3.1 Regression

We begin with 1-dimensional Bayesian regression. 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.

3.1.1 Linear regression. We start with a non-lazy model: linear regression. Following the Bayesian tradition, 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.

Our prior is as follows. The slope `a` and intercept `b` are both drawn from normal distributions.

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

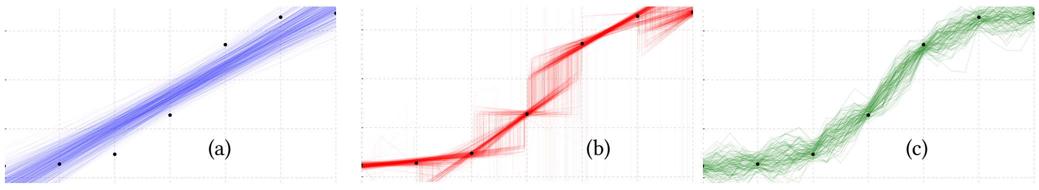


Fig. 3. Bayesian regression in LazyPPL for the data set indicated by the dots. We illustrate the posteriors starting from three different priors on the function space. From left to right: (a) linear (§3.1.1), (b) piecewise linear (§3.1.2), and Wiener (see §6.4).

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. To this end we define a general purpose function `regress`, which takes a standard deviation `sigma`, a prior over the function space `prior`, and a list of `(x,y)` observations `dataset`.

```
regress :: RealNum → Prob (a → RealNum) → [(a,RealNum)]
                                          → Meas (a → RealNum)
regress sigma prior dataset =
  do f ← sample prior
     forM dataset $ \(x,y) → score $ normalPdf (f x) sigma y
     return f
```

So linear regression in particular is achieved by `regress 0.1 linear dataset`, see Figure 3 (a).

3.1.2 Piecewise regression. The function `regress` can be used for more involved kinds of regression. We consider *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:

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

The sequence of change points can be infinite, so the function can have an infinite number of pieces, but this is handled lazily.

In general a random collection of points is called a *point process.* A simple example of an infinite point process is a *Poisson* point process on the positive reals, generated by repeatedly sampling steps from an exponential distribution with a fixed rate (see also Figure 1 (a)):

```
poissonPP :: RealNum → RealNum → Prob [RealNum]
poissonPP lower rate = do step ← exponential rate
                          let x = lower + step
                          xs ← poissonPP x rate
                          return (x : xs)
```

We can perform piecewise linear regression using a Poisson point process, for example via `regress 0.1 (splice (poissonPP 0 0.2) linear) dataset`.

Key point of laziness. Notice that the probability measure `splice (poissonPP 0 0.2) linear` has infinite dimension, because the change points continue across the real line. But LazyPPL has no problem with this. Intuitively, it is all fine because the data set is finite and the plotting routine only inspects a finite viewport, and so the built-in laziness avoids any infinite computation.

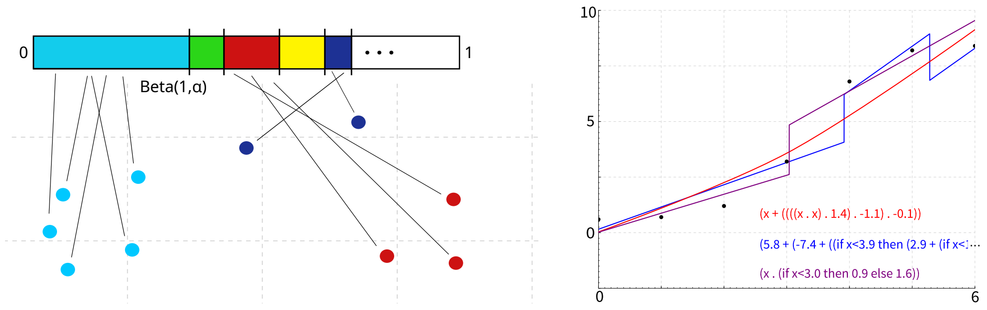


Fig. 4. From left to right: (a) Dirichlet process clustering by stick-breaking in LazyPPL; (b) Program induction via Bayesian regression in LazyPPL (§6.2) (the second inferred program is abbreviated since it is quite long)

As an aside, we point out the beauty of probabilistic programming for modular 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.

3.2 Clustering

We now illustrate clustering in LazyPPL. For a set of data points, clustering is the problem of finding the most appropriate partition into clusters. Taking advantage of laziness, we can allow for an unbounded number of clusters.

3.2.1 Stick-breaking. In *Bayesian* clustering, we have a prior distribution over possible partitions of the data into clusters. We implement this using stick-breaking, where we break the unit interval $[0,1]$ into an infinite number of sticks, each representing a cluster, and the size of the stick is the proportion of points in that cluster. Stick-breaking is easy to define lazily. At each step we break off a portion of the remaining interval according to a beta distribution:

```
stickBreaking :: RealNum → RealNum → Prob [RealNum]
stickBreaking α lower =
  do r ← beta 1 α
     let v = r * (1 - lower)
     vs ← stickBreaking α (v + lower)
     return (v : vs)
```

Given the broken sticks, we can organise the data into clusters. For every data point, we uniformly draw a value from $[0,1]$ and use the corresponding stick as a cluster assignment (Fig. 4). Thus we have a prior over cluster assignments for any data set:

```
assignClusters :: RealNum → [a] → Prob [(a, Int)]
```

We then need to incorporate the likelihood of a particular clustering. One way is to assign a mean value to each cluster, and assume that the data points in this cluster follow a normal distribution around that mean. More abstractly, we define:

```
cluster :: [a] → (Prob b) → (b → a → RealNum) → Meas [(a, Int)]
```

Our clustering model has the form (`cluster dataset base likelihood`). The program 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`.

3.2.2 *Stochastic memoization.*

The previous clustering example (§3.2.1) can be implemented with stochastic memoization. In standard programming, memoization is a form of laziness where a function caches previous results instead of re-calculating [42]. In functional probabilistic programming, memoization also becomes a powerful method for building infinite-dimensional probability measures (e.g. [24, 54, 77]). In LazyPPL, we consider a *typed* memoization function

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

As the types make clear, `memoize` converts a parameterized distribution `p` into a random function (`memoize p`), informally by sampling once from (`p x`) for every (`x :: a`). Memoize can often be defined in terms of laziness. For example, for positive integers, we can define

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

In practice, we actually define this more efficiently using tries (c.f. [30]). Not every type `a` supports memoization, and we define a type class `MonadMemo` for those that do.

In the clustering example, we can use memoization to assign a mean to each cluster. One implementation of our clustering model above is:

```
cluster dataset base likelihood = do
   assignment ← sample $ assignClusters 0.3 dataset
   means ← sample $ memoize $ \c → base -- lazily map values to clusters
   mapM (\(x, c) → score $ likelihood (means c) x) assignment
   return assignment
```

The memoized function `means` is an infinite-dimensional object. There is no bound on the number of clusters, and LazyPPL computes only the dimensions we need.

3.2.3 *Chinese restaurant process and abstract data types.*

The random partition of the data set arising from the stick-breaking construction above is called a *Chinese restaurant process* (CRP) (e.g. [21]). The idea is to think of data points as customers arriving into a restaurant. Each customer is assigned a table, and so tables correspond to clusters.

The stick-breaking construction is only one possible implementation of the CRP. There is also no requirement that tables should be represented as integers. We find it useful to encapsulate our representations using abstract data types:

```
newtype Restaurant = R [RealNum]
newtype Table      = T Int deriving (Eq,MonadMemo Prob)
```

These come with constructors:

```
newRestaurant :: RealNum → Prob Restaurant
newCustomer   :: Restaurant → Prob Table
```

For our specific implementation, (`newRestaurant` α) just performs the stick-breaking, and (`newCustomer r`) draws a random stick with probability equal to its length.

This abstract interface prevents the programmer from accessing the underlying representation. This is important, because we want to remove the artificial ordering on clusters to preserve *statistical exchangeability*: the idea that data points can be considered in any order and the distribution of the partition does not change (e.g. [65, 66]). Although the implementation of `Table` is hidden, notice that we retain the fact that it supports memoization.

With this interface, the cluster assignment function has a clear implementation:

```
assignClusters :: RealNum → [a] → Prob [(a, Table)]
assignClusters α dataset = do
   r ← newRestaurant α
   mapM (\x → (x, newCustomer r)) dataset
```

Regression and clustering are widely studied problems, but we already see the power of laziness and types at this level. In LazyPPL we can seamlessly manipulate complex or infinite-dimensional distributions. We will explore this with more advanced examples in Section 6.

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

4 IMPLEMENTATION OF THE PROBABILITY MONAD WITH INFINITE TREES

We now introduce and motivate our implementation of probability.

Let Ω be a set of random seeds. A *randomized function* between sets X and Y is a function $f\colon X \times \Omega \to Y$, that depends on the random seed. Suppose that we have a method for splitting random seeds, $\gamma : \Omega \to \Omega \times \Omega$ (e.g. [67]). Then we can compose randomized functions

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

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

By currying, we can regard a function $f\colon X \times \Omega \to Y$ as a function $X \to Y^{\Omega}$. Once we equate certain functions, Y^{Ω} becomes a monad, and we are thus in the setting of programming with monads [43].

We now recall the rudiments of measure theory (§4.1), and then formalize associativity of composition (§4.2). We also treat unnormalized measures (§4.3).

4.1 Rudiments of quasi-Borel spaces

In Section 2.2 we recalled the notions of measurable space and measure. As we noted, this does not support function spaces, nor is it known to support commutative monads of measures. We now recall a setting that does support both of these things: quasi-Borel spaces.

To recall the definitions, we need to recall the notion of *standard Borel space*. In fact, we do not need the traditional definition; the following characterization will suffice.

Proposition 4.1 (e.g. [33]). (1) *A standard Borel space is a measurable space (X, Σ_X) that is either (a) countable, with Σ_X the powerset of X, or (b) measurably isomorphic to $(\mathbb{R}, \Sigma_{\mathbb{R}})$.*
(2) *Any measurable subspace of $\mathbb{R}$ is standard Borel (e.g. $[0, 1]$ is standard Borel).*
(3) *Standard Borel spaces are closed under countable products.*

Following our introductory discussion, let Ω be a fixed uncountable standard Borel space (typically $\Omega = \mathbb{R}$, but we also consider $\Omega = [0, 1]^{\mathbb{N}^*}$). We now recall:

Definition 4.2 ([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 $f \colon \Omega \to \Omega$ is measurable then $(\alpha \circ f) \in M_X$;
- gluing: if $\alpha_1 \dots \alpha_n \dots \in M_X$ and $\Omega = \biguplus_{n=1}^{\infty} U_i$ measurable then $\alpha \in M_X$ where $\alpha(\omega) = \alpha_n(\omega)$ where $\omega \in U_i$.

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

Proposition 4.3 ([29]). *Quasi-Borel spaces and functions form a category* **Qbs** *that is cartesian closed. Standard Borel spaces* (X, Σ_X) *fully embed in quasi-Borel spaces, taking* M_X *to be the measurable functions.*

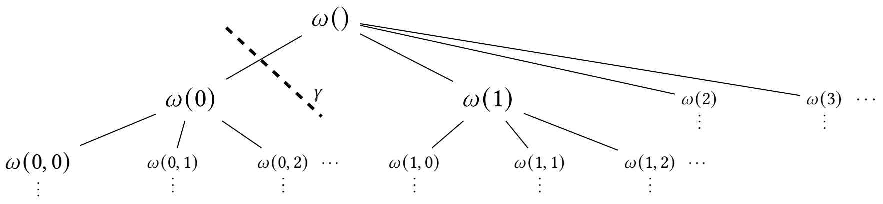
Thus, we can understand the types of a functional programming language as quasi-Borel spaces. In the introductory paragraph to this section, X and Y should be regarded as quasi-Borel spaces and the functions f, g as quasi-Borel functions.

4.2 A category of probability kernels

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

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

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



Our probability measure μ on this choice of Ω is the countably-infinite product measure of the uniform distribution, given by the Kolmogorov extension theorem. For each path $(i_1, \dots, i_n) \in \mathbb{N}^*$, 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 $\upsilon : \Omega \to [0, 1]$.

4.2.2 Probability kernels.

Definition 4.4. Let X and Y be quasi-Borel spaces. A *probability kernel* $f \colon X \rightsquigarrow Y$ is a quasi-Borel function $f \colon X \times \Omega \to Y$. We consider the equivalence relation on probability kernels that is determined by

$$f \sim g \text{ if for all } x \in X \text{ and all morphisms } h : Y \to \mathbb{R}, \int h(f(x, \omega))\, \mu(\mathrm{d}\omega) = \int h(g(x, \omega))\, \mu(\mathrm{d}\omega). \tag{3}$$

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 $v : \Omega \to [0,1]$.
- For any X, the identity probability kernel $X \rightsquigarrow X$ is the projection function $X \times \Omega \to X$.
- We *compose* two probability kernels $f\colon X \rightsquigarrow Y$, $g\colon Y \rightsquigarrow Z$, obtaining a probability kernel $gf\colon X \rightsquigarrow Z$ given by:

$$X \times \Omega \xrightarrow{X \times \gamma} X \times \Omega \times \Omega \xrightarrow{f \times \Omega} Y \times \Omega \xrightarrow{g} Z$$

- We *tensor* two probability kernels $f\colon A \rightsquigarrow B$, $g\colon X \rightsquigarrow Y$, obtaining a probability kernel $f \otimes g\colon (A \times X) \rightsquigarrow (B \times Y)$ given by:

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

Proposition 4.5. *Probability kernels modulo equivalence form a monoidal category* **ProbKer**: *that is, composition and tensor are associative and unital up to equivalence, the interchange law is satisfied up to equivalence, and the operations on probability kernels respect equivalence relations.*

We can regard any quasi-Borel function $X \to Y$ as a probability kernel $X \times \Omega \to X \to Y$; this induces an identity-on-objects functor **Qbs** $\to$ **ProbKer**.

Proposition 4.6 (see [29]). *The inclusion functor* **Qbs** $\to$ **ProbKer** *has a right adjoint. That is, the functions $(\Omega \to X)$ modulo equivalence form a monad on the category of quasi-Borel spaces.*

4.2.3 Aside on alternative representations. We note a different notion of randomized function, where the function is equipped with a parameter space (following e.g. [62]; see also [40, 61]). Let us briefly define a *para-randomized* function between sets X and Y to be a pair (Ω, f), where Ω is a standard probability space and $f\colon X \times \Omega \to Y$ is a function. Unlike with our randomized functions, the seed space is not fixed and is part of the data for a para-randomized function. Composition is of the form

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

with $(g \circ f)(x, (\omega_1, \omega_2)) = g(f(x, \omega_1), \omega_2)$. This formulation is convenient when a function has a natural parameter space of fixed dimension such as $\mathbb{R}^3$. Then composing $(f, \mathbb{R}^m)$ and $(g, \mathbb{R}^n)$ yields $(g \circ f, \mathbb{R}^{m+n})$. This is reasonable for certain simple probabilistic programs, but in this article we are especially interested in the situation where the parameter spaces are not so simple. For example, in §3.1.2 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.

To achieve a proper monoidal category of para-randomized functions, it is necessary to quotient, otherwise the interchange laws and associativity fail. In this case we can quotient again by (3). In fact, this yields a category isomorphic to **ProbKer**. But the isomorphism is difficult to compute with in practice, and so we will not discuss this formulation further in what follows.

4.3 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 Fig 2 (d). Densities can also construct unnormalized measures: starting from the standard normal distribution on $\mathbb{R}$, the weight $(\sqrt{2\pi})e^{\frac{1}{2}x^2}$ defines the Lebesgue measure on $\mathbb{R}$, which assigns to each open interval its length (see also [64]).

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 probabilistic programming, `sample` and `score`.

Definition 4.7. A *measure kernel* $(f, \ell)\colon X \rightsquigarrow Y$ is a pair of quasi-Borel functions, $f : X \times \Omega \to Y$, $\ell\colon X \times \Omega \to [0, \infty]$. We consider the equivalence relation on probability kernels that is determined by $(f, \ell) \sim (f', \ell')$ if for all $x \in X$ and all morphisms $g : Y \to \mathbb{R}$,

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

We can perform various constructions on measure kernels too.

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

Proposition 4.8 ([61], §4.3.3). *Measure kernels modulo equivalence form a monoidal category. The inclusion functor* **Qbs** $\to$ **MeasKer** *has a right adjoint, and so the functions* $\Omega \to (X \times \mathbb{R})$ *modulo equivalence form a monad on the category of quasi-Borel spaces.*

Note. Although this appears similar to the writer monad transformer applied to the probability monad, in fact the equivalence relation is coarser than this, because sometimes different weights give rise to the same notion of integration.

4.4 Implementation of the probability monads

It is difficult to work with equivalence relations in practical functional programming. To avoid this, we implement the above without equivalence relations but work behind a module abstraction to avoid distinguishing equivalent terms.

We first define rose trees, with γ = `splitTree`:

```
data Tree = Tree Double [Tree]
splitTree :: Tree → (Tree , Tree)
splitTree (Tree r (t : ts)) = (t , Tree r ts)
```

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

```
newtype Prob a = Prob (Tree → a)
uniform :: Prob Double
uniform = Prob $ \(Tree r _) → r
return a = Prob $ const a
(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. In fact, a similar bind is used in QuickCheck [12, §6.4], although we are not aware of a worked out semantic argument in the prior literature.

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

```
newtype Meas a = Meas (WriterT (Product (Log Double)) Prob a)
score :: Double → Meas ()
score r = Meas $ tell $ Product $ (Exp . log) $ r
sample :: Prob a → Meas a
sample p = Meas $ lift p
```

4.4.1 Basic inference: Likelihood weighted importance sampling. The reference inference method is likelihood weighted importance sampling. This is implemented in LazyPPL as a function

```
lwis :: Int → Meas a → IO [a]
```

which uses the following pseudocode, for (`lwis` n `m`):

(1) generate n pairs of weighted samples from `m`, i.e. pairs (w_i, x_i) of a weight w_i and result x_i in `a`, each according to the following algorithm:
  (a) lazily initialize a random rose tree `t :: Tree`, by allocating a fresh random number to every node (this is where the `IO` monad is used, although a different probability monad could be used instead;
  (b) run the program `m :: Meas a` with the rose tree `t`;
  (c) return the pair (w, x) of the resulting value and the accumulated weight.
(2) generate an infinite stream of samples from the discrete 'empirical' distribution $\sum_{i=1}^{n} \frac{w_i}{\sum_{i=1}^{n} w_i} \cdot x_i$, as follows:
  (a) pick a random number r uniformly in the interval $[0, \sum_{i=1}^{n} w_i]$;
  (b) if r is in the interval $[\sum_{i=1}^{j-1} w_i, \sum_{i=1}^{j} w_j]$ then output x_j.

As $n \longrightarrow \infty$, the empirical distribution almost surely converges to the normalized probability distribution. So this is an approximate sampler from the normalized probability distribution corresponding to the unnormalized measure `m`. But in practice, for small n, it is not usually very good, and so we look at a better algorithm in Section 5.3.

5 METROPOLIS-HASTINGS SIMULATION

Recall that a closed probabilistic program induces a pair of functions $\mathbb{R} \xleftarrow{\ell} \Omega \xrightarrow{f} X$, where Ω is regarded with a basic probability measure p, and ℓ is measurable. Here ℓ is regarded as a density for an unnormalized distribution on Ω, which is then to be pushed forward to X, which is the space of interest. There are four measures of interest:

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

  `do {w ← sample mu ; score (l w); return w} :: Meas Omega`.

  (Here, and throughout Section 5.1 and the proof of Thm. 5.2, we are using LazyPPL syntax to discuss and manipulate semantic measures, in the spirit of synthetic measure theory – these are not necessarily programs to be run directly.)
- The *normalized* form of the measure μ_ℓ, $\frac{\mu_\ell}{\mu_\ell(\Omega)}$, which is a probability measure, assuming $\mu_\ell(\Omega) \in (0, \infty)$.
- The pushforward probability measure on X, $f^*(\frac{\mu_\ell}{\mu_\ell(\Omega)})$. This could be written

`do {w ← sample mu ; score (l w); score (1/(mu`$_\ell$ `Omega)) ; return (f w)} :: Meas X`.

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

5.1 Proposal kernels in general

The key ingredient for a Metropolis-Hastings algorithm is a 'proposal' Markov kernel. This is a function $k : \Omega \times \Sigma_\Omega \to [0,1]$ such that each $k(\omega,-)$ is a probability measure and each $k(-,U)$ is measurable. We follow the analysis of proposal kernels from [20, 27].

The proposal kernel k does not directly capture the probability measure $\frac{\mu_\ell}{\mu_\ell(\Omega)}$. Rather, it induces another kernel, which works by first proposing changes (using k) and then either accepting or rejecting them (§5.3).

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

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

These can be described in programming terms as

```
    m = do {w₁← sample mu ; w₂← sample (k w₁) ; score (l w₁) ; return (w₁,w₂)}
m_rev = do {w₁← sample mu ; w₂← sample (k w₁) ; score (l w₁) ; return (w₂,w₁)}
```

Definition 5.1 ([27]). We say that a kernel k is *Green* with respect to ℓ and μ if m_{rev} is absolutely continuous with respect to m. This means that there exists a 'ratio' $r : \Omega \times \Omega \to \mathbb{R}$ (the 'Radon-Nikodym derivative') such that

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

or in programming terms

```
  do {w₁← sample mu; w₂← sample (k w₁); score (l w₁); score (r w₁ w₂); return (w₁,w₂)}
   = do {w₁← sample mu; w₂← sample (k w₁); score (l w₁); return (w₂,w₁)}
```

5.2 A new proposal kernel for lazy rose trees

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

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

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

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

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

Theorem 5.2. *The kernel $k : \Omega \times \Sigma_\Omega \longrightarrow [0,1]$ given by* `(mutateTree p)` *is Green, and the ratio is* $r(\omega_1, \omega_2) = \frac{\ell(\omega_2)}{\ell(\omega_1)}$.

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

```
do {w₁← mu; w₂← k w₁; return (w₁,w₂)}  =  do {w₁← mu; w₂← k w₁; return (w₂,w₁)}
```

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

```
do {w₁← sample mu; w₂← sample (k w₁); score (l w₁); score (r w₁ w₂); return (w₁,w₂)}
 = do {w₁← sample mu; w₂← sample (k w₁); score (l w₂); return (w₁,w₂)}
 = do {w₁← sample mu; w₂← sample (k w₁); score (l w₁); return (w₂,w₁)}
```

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

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

5.3 The Metropolis-Hastings-Green Markov Chain

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

```
kMHG :: Omega → Prob Omega
kMHG w₁ = do
  w₂ ← k w₁
  b ← bernoulli $ min 1 (r w₁ w₂)
  if b then return w₂ else return w₁
```

We can then construct a Markov chain with transitions given by k_{MHG}. The key result (e.g. [20, 27]) is that when k is well-behaved, the states of this Markov chain approximate the posterior distribution. Theorem 5.3 says this formally. Recall that a probability measure ν on Ω is a stationary distribution for a kernel $k : \Omega \times \Sigma_\Omega \longrightarrow [0,1]$ if

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

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

Theorem 5.3 (Metropolis-Hastings-Green). *For any Green kernel k, the induced kernel k_{MHG} has a stationary distribution, which is the normalized probability measure $\frac{\mu_\ell}{\mu_\ell(\Omega)}$ on Ω. If k_{MHG} is irreducible with respect to $\frac{\mu_\ell}{\mu_\ell(\Omega)}$ then the stationary distribution is unique.*

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

Proposition 5.4. *For the* `mutateTree` *kernel (§5.2) with $p = 1$, k_{MHG} is irreducible for $\frac{\mu_\ell}{\mu_\ell(\Omega)}$.*

Proof note. Here $n = 1$ suffices. □

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

There remains a concern that k_{MHG} is not irreducible for $p < 1$. Indeed, in that situation, the set $U = \{\omega' \mid \forall i.\, \omega_i \neq \omega'_i\}$ is not reachable from ω, even though U typically has measure 1. More informally, although every node has a chance of being changed, there will almost surely exist a node that is not changed. One way to resolve this would be to mix (`mutateTree p`) with (`mutateTree 1`), using Theorem 7.1; the resulting kernel is trivially irreducible, even if the chance of using (`mutateTree 1`) is kept miniscule. In practice, (`mutateTree p`) alone appears to be fine, because any finite collection of samples will only invoke a finite number of nodes anyway.

5.4 Summary and example

In summary, we have a procedure for sampling from the distribution described by a probabilisitic program, by using the Metropolis-Hastings-Green kernel (§5.3) associated to the Green Markov kernel (`mutateTree p`) (§5.2). Each step of the algorithm provides a sample from the measure $\frac{\mu_\ell}{\mu_\ell(\Omega)}$ on Ω, and we can push-forward this sample along $f\colon \Omega \to X$ to obtain a sample from the measure described by the probabilistic program.

To illustrate, we recall the simple linear regression model (§3.1). 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.

6 MORE ADVANCED ILLUSTRATIONS

We demonstrate the power of laziness for constructing models, illustrating more examples from the theory of Bayesian non-parametrics and practical probabilistic programming.

6.1 Laziness, control flow and addressing

We consider a very simple model, to illustrate the kind of reorganization that laziness allows. Suppose that we toss a fair coin, and then, depending on the outcome, we toss one of two biased coins. We then notice that we got the same result both times. What was the result?

```
model :: Prob (Bool, Bool)
model = do x ← bernoulli 0.5
           y ← if x then bernoulli 0.4 else bernoulli 0.7
           return (x,y)
test = do {(x,y) ← sample model; score (if x==y then 1 else 0); return x}
```

The result is `True` with probability $\frac{0.5\times0.4}{0.5\times0.7} \approx 0.57$. Because the probability monad is affine, the model is equivalent to the situation where we actually tossed all three coins: by laziness, only two coins will only ever actually be looked at on any run.

```
model2 = do x ← bernoulli 0.5
            ytrue ← bernoulli 0.4
            yfalse ← bernoulli 0.7
            return (if x then (x, ytrue) else (x, yfalse))
```

Although the second formulation might look more costly, in fact the run time cost is roughly the same, because of laziness, and moreover, it results in fewer rejections with our new Metropolis-Hastings kernel (§5.2). This is because in the second formulation, all three bernoulli statements are allocated different nodes in the rose tree, and so the proposal will often be accepted even if just `x` changes. In the first formulation, however, the proposal will be rejected unless both `x` and `y` are changed simultaneously. These kinds of 'addressing' issues have been noted by other authors (e.g [34]), and we note that laziness provides a clear way to deal with them.

6.2 Regression with a prior over program expressions

We revisit the regression scenario from Section 3.1. We can phrase program induction as a regression problem. This is done in e.g. [73], but we discuss it here because the typed setting is clarifying, and it illustrates some further aspects of laziness. We consider a prior over program expressions, `randExpr :: Prob Expr`, an evaluation function `Expr → RealNum → RealNum`, and then calling

```
regress sigma dataset (do { e ← randExpr ; return (eval e) })
```

We used this method to infer the expressions shown in Figure 3. In more detail, but keeping things fairly simple, we consider a datatype

```
data Expr = Var | Constt RealNum | Add Expr Expr | Mult Expr Expr
          | IfLess RealNum Expr Expr
```

for expressions with a single variable, and consider an evaluation function `Expr → RealNum → RealNum` defined by induction over expressions. It remains for us to discuss our prior, which is a random expression built by recursively making choices about what its subexpressions will be. Here we use the method of Section 6.1, exploiting laziness to simultaneously consider all the possible expression choices, which are now infinite.

```
randExpr :: Prob Expr
randExpr = do
  i ← categorical [0.3,0.3,0.18,0.18,0.04]
  es ← sequence
     [return Var ,
     do { n ← normal 0 5 ; return $ Constt n },
     do { e1 ← randExpr ; e2 ← randExpr ; return $ Add e1 e2},
     do { e1 ← randExpr ; e2 ← randExpr ; return $ Mult e1 e2},
     do { r ← normal 0 5 ; e1 ← randExpr ; e2 ← randExpr ;
          return $ IfLess r e1 e2}]
  return $ es !! i
```

When we run inference, each expression template (roughly a 'sketch', [63]) is allocated a different path through the rose tree. Some likely results are shown in Fig. 4(b).

6.3 General stochastic memoization

In Section 3.2.2 we discussed stochastic memoization, a second-order function `memoize`, which is a crucial primitive in Church [24]. Memoization is a form of laziness, and conversely, as we discussed, it can be defined in using pure laziness for many specific types. An alternative approach is to actually use a memo table, and then we can define `memoize` whenever its argument supports equality testing. We define

```
generalMemoize :: Eq a => (a → Prob b) → Prob (a → b)
```

Our implementation ensures that a sample from `generalMemoize f` returns a function that has a fresh hidden memo table of pairs `(x,y)` of argument/result pairs, initialized empty. When the function is subsequently called with argument `x`:

- If a value at `x` is already stored in the memo table, return that value.
- If not, `x` sample `y ← f x`, store `(x,y)` in the memo table, and return `y`.

To work with this memo table, we need to use state. (We do this via `unsafePerformIO`.) Our implementation is safe in the following sense:

- the sampled function appears deterministic to the user: function calls can be reordered, discarded, and copied [19]; and
- the resulting distribution (`generalMemoize f`) in (`Prob (a → b)`) satisfies the commutativity and affine laws (1, 2).

As a simple example, we can define white noise via

```
generalMemoize (\x → uniform) :: Prob (RealNum → RealNum)
```

White noise is difficult to analyze in measure-theoretic terms, and it is curious that this coincides with the limitations of pure laziness in functional programming.

6.4 Regression with a Gaussian process

Gaussian processes (GPs) are examples of random real-valued functions, so their type in LazyPPL is `Prob (a → RealNum)`. To illustrate the key points, we have focused on a very simple Gaussian process, the one-dimensional Wiener process, also known as Brownian motion. This has type `Prob (RealNum → RealNum)`.

A draw from the Wiener process is a function that is almost surely continuous everywhere but differentiable nowhere. Informally, the Wiener process is the continuous-time version of a symmetric random walk on $\mathbb{R}$, starting at 0. We have used this as a prior for a regression model, using the `regress` function and the dataset from Section 3.1. See Figure 3 (right) for some draws from the posterior.

Although the Wiener process formally involves an uncountable number of random choices, in practice for Wiener process regression we only need to know its value at the points where it is evaluated. These depend on the observation points, the resolution of the plot, and the viewport. Thus although we use the Wiener process as a random function, it makes practical sense because it can be evaluated lazily.

The Wiener process challenges our primitives for laziness. Before turning to our implementation of the Wiener process, we first discuss a simpler discrete Gaussian random walk, `Prob (Int → RealNum)`. At each step, this moves according to a normal distribution. This might be defined as

```
rw :: Prob (Int → RealNum)
rw = do { ys ← iterateM (\y → normal y 1) 0 ; return (\x → ys !! x) }
```

Suppose we subsequently evaluate the position at 1 then 5 and then 3:

```
do {f ← rw ; (f 1 , f 5, f 3)}
```

This implementation is lazy – it will never actually make the random choices needed to determined the position at time 6. But it will need to calculate the position at times 2 and 4. In fact, it is possible to avoid these by following the *Brownian bridge* method (e.g. [23, §3.1.1, Fig. 3.1]):

```
do f₁ ← normal 0 1
   f₅ ← normal 0 (sqrt 2)
   f₃ ← normal (f₁ + 0.5*(f₅-f₁)) 1
   return (f₁ , f₅, f₃)
```

We can also implement the Brownian bridge method without knowing in advance which arguments a function will be called with. Somewhat similarly to §6.3, our implementation is to return a function that has a hidden memo table of pairs `(x,y)` of argument/result pairs, initialized with one entry `(0,0)`. When the function is subsequently called with argument `x`:

- If a value at `x` is already stored in the memo table, return that value.
- If not, and if `x` is lower than the lowest entry `(x₀, y₀)`, or higher than the highest entry `(x₀, y₀)`, sample `y ← normal y₀ (sqrt |x₀ - x|)`, store `(x, y)` in the memo table, and return `y`.
- Otherwise, if `(x₀, y₀)` and `(x₁, y₁)` are the nearest points in the memo table ($x_0 \leq x \leq x_1$), then sample `y ← normal (y₀ + r*(y₁ - y₀)) (sqrt (r * (x₁ - x)))`, and store `(x, y)` in the memo table, and return `y`. Here `r = (x - x₀)/(x₁ - x₀)`.

To store the memo table, we need to use state (via `unsafePerformIO`). Our implementation is safe in the same sense as the stateful stochastic memoization (§6.3): the sampled function appears deterministic to the user, and the resulting distribution `Prob (Int → RealNum)` satisfies the commutativity and discardability laws.

This exact same pseudocode works for the Wiener process, and this is how we implement `wiener :: Prob (RealNum → RealNum)`. In the case of functions from the integers, this Brownian bridge implementation is more efficient than the list based implementation `rw`. But in the Wiener process, which takes real valued arguments, there appears to be no other way to implement this random function. Thus 'continuous' stochastic processes, which push against the measure-theoretic foundations of probability, also push against the traditional lazy foundations of functional programming.

6.5 Feature extraction and the Indian Buffet Process

A *feature assignment* for a data set is 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. We can represent a feature assignment as a boolean-valued matrix, where the columns are features and the rows are data points.

We consider the problem of feature extraction, where we have a data set but the features are unknown. We can view this as a generalized version of the clustering problem where clusters are allowed to overlap, so a data point could belong to many clusters.

The Indian Buffet Process (IBP) is a Bayesian model for feature extraction [28]. It provides a distribution on boolean-valued matrices, described by a process for sampling rows one by one. As with the CRP, we think of data points as customers walking into a restaurant, but here each customer selects a number of dishes (features) from a buffet. The dishes are selected based on their popularity with previous customers. To implement the IBP we use some abstract types:

```
newtype Restaurant  = R ([[Bool]], Counter)
newtype Dish        = D Int  deriving (Eq, MonadMemo Prob)
newRestaurant :: RealNum → Prob Restaurant
newCustomer   :: Restaurant → Prob [Dish]
```

The idea is that a restaurant is initialized with an infinite matrix describing an infinite (but lazy) simulation of the process. The counter keeps track of how many rows have already been consumed by customers coming in. A call to (`newCustomer r`) will simply look up the next row in the matrix, return the column indices containing 1's, and increment the counter in `r`.

A probabilistic counter. The `Counter` type is introduced to encapsulate an integer reference:

```
data Counter = C (IORef Int)
```

The main reason for this encapsulation is to hide and restrict any stateful operations in the modelling interface. Although our use of state is safe in practice, the counter will affect the distributions, so all operations are probabilistic:

```
newCounter        :: Prob Counter
readAndIncrement :: Counter → Prob Int
```

It is notable that the IBP seems to necessitate some deviation from pure laziness in its implementation, which appears to coincide with continuity issues in the analysis of the IBP (e.g. [56, Sl. 6], also [1, 54, 55]). (We note that a counter can be useful for other models, beyond the IBP, to keep track of the process history. One example is an alternative representation of the Chinese Restaurant Process in which a new customer is assigned a table based on the locations of previous customers.)

A feature extraction example. We turn to an application of the IBP. We have used LazyPPL to solve a basic problem from applied psychology [49]. The problem is as follows. We have a set of 16 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. In the Bayesian model we consider, the similarity coefficients are calculated from the subset of features that two countries share. We use an IBP prior on feature assignments for the set of countries, and incorporate the experimental data to infer a distribution on probable feature assignments.

Our results are similar to those in [49], surprisingly so since we just used the plain inference of Section 5. The features are inferred and abstract, but they are interpretable. For example, in Fig. 5, columns 1 and 5 could correspond to Asia and Caribbean respectively; column 7 could indicate conflicts.

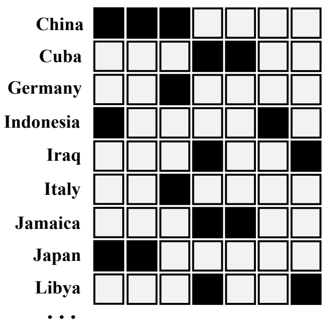


Fig. 5. Feature extraction for a psychological experiment. The columns correspond to features and assignments are automatically inferred using an IBP model.

6.6 Relational modelling and Mondrian process

As a final example, we consider the Mondrian process [57], a model often used to generate random relations between sets. Formally, a draw from a Mondrian process is a block partition of a k-dimensional hypercube of the form $\Theta_1 \times \cdots \times \Theta_k$, where each Θ_d is a closed interval $[a_d, b_d]$. (see also [3]). These are easy to manipulate with Haskell data types:

```
data Mondrian a = Block a [(RealNum, RealNum)]
    | Partition Int RealNum [(RealNum, RealNum)] (Mondrian a) (Mondrian a)
```

Our implementation of the Mondrian process provides a parameterized sampler:

```
newMondrian :: (Prob a) → RealNum
                           → [(RealNum, RealNum)] → Prob (Mondrian a)
```

A call to (`newMondrian` μ λ Θ) follows a recursive process, which we summarize now.

The intuitive idea is that we make one cut through the hypercube Θ, and then recursively call the Mondrian process on each separate part. The parameter $\lambda \in \mathbb{R}$ is a fixed budget, which determines how deep the recursion will go. Every cut takes away some amount from the budget, sampled from an exponential distribution, and we stop when the budget has run out. The direction of the cut is orthogonal to one of the axes, chosen at random with probability proportional to the length of the corresponding interval $[a_d, b_d]$. Then, we draw a point uniformly in $[a_d, b_d]$ to determine the cut position. This yields a k-d tree, whose leaves are k-dimensional blocks partitioning the initial domain Θ (the root of the tree) and whose internal nodes are given by a cut point $x_d \in \mathbb{R}$ splitting a subdomain $\Theta' \subseteq \Theta$ across a given axis $d \in \mathbb{N}$ (*cf.* the `Mondrian` data type). Interestingly, in dimension $k = 1$ the positions of the cuts form a Poisson point process (§1.2).

After generating the partition, we annotate each block with a sample p from the parameter distribution μ. Typically, μ is a distribution on $[0, 1]$ so that p represents the probability that points associated with this block should be related. (In two dimensions and for $\Theta = [0, 1]^2$, we have a particular kind of *graphon*, see e.g. [50].)

Experiments with a generative model for relational data. Following [57], we have experimented with Mondrian process inference using a synthetic data set, consisting of relational data generated from an actual 1921 painting by Piet Mondrian (Figure 6 (a,b)). This is intended as a simple proof of concept. The implementation relies on a function for generating relations lazily:

```
sampleRelationFromMondrian2D :: Mondrian RealNum → Prob Relation
```

`Relation` is a type of lazy infinite binary relations, or matrices. We define abstract types `Row` and `Col` and obtain an interface for modelling with exchangeable arrays:

```
data Relation = Relation Counter Counter [[Bool]]
newRow :: Relation → Prob Row
newCol :: Relation → Prob Col
lookup :: Relation → Row → Col → Bool
```

An example posterior sample is displayed in Figure 6 (c). We can experiment with variants of this model. For instance, if we record the plane coordinates when creating our synthetic data points, our inference algorithm recovers block partitions analogous to the initial painting (Figure 6 (d)).

6.7 Summary

Probabilistic programming makes it easy to construct hierarchies or combinations of models. This is especially true in LazyPPL where the types and lazy data structures make the modularity clear.

We emphasize that laziness can be exploited at every level: for instance it is easy to generate a Mondrian relation over the tables of a Chinese Restaurant for a dataset, even without knowing in advance how many tables there are. There are many practical examples of this kind of construction: Mondrian Forests [?], Chinese Restaurant Franchises [5], hierarchical Generalized IBPs [55], and others. For these complex models it is harder to reason about symmetries and exchangeability, so abstract types for encapsulating internal details are another benefit of LazyPPL.

7 SINGLE-SITE METROPOLIS-HASTINGS WITH GHC-HEAP

The inference algorithm in Section 5 randomly mutates each node (also sometimes referred to as *site*) in the lazy rose tree with some given bias p at every proposal step. Such an inference technique may not always be desirable, however, e.g. in situations where we want to control the exact number of nodes being mutated in each proposal (so that our mutations are more gradual): we can never guarantee a constant fixed number of mutations due to them happening independently of each other.

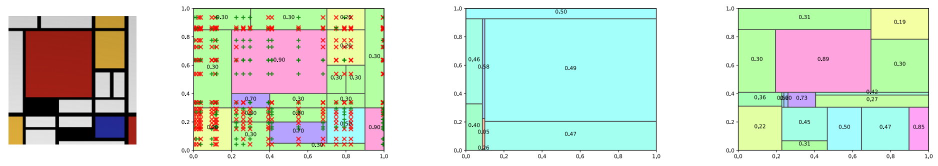


Fig. 6. Left to right: (a) "*Composition with Large Blue Plane, Red, Black, Yellow, and Gray*" by Mondrian (1921), (b) our representation of (a), and a random synthetic relation (red = `false` for a pair of points, green = `true`), (c) a posterior sample, and (d) a variant of the experiment recording the plane coordinates.

Wingate et al. [76] suggest a proposal kernel where only a single node is mutated at every proposal step. In this Section we discuss our own implementation of this in LazyPPL. We found this to be particularly profitable where it is unclear which p to use, as in the clustering example (§3.2).

7.1 State-dependent mixing in general

We consider the following general method for mixing Green kernels (Def. 5.1), which is perhaps implicit in [20, 27]. Let $k_i : \Omega \times \Sigma_\Omega \to [0, 1]$ be a countable family of Markov kernels (§5.1), and let $c \colon \Omega \times \mathbb{N} \to [0, 1]$ be a parameterized probability distribution function over $\mathbb{N}$, i.e. for all $\omega \in \Omega$, $\sum_{i=1}^{\infty} c(\omega, i) = 1$. Let k be the mixed kernel

$$k(\omega, U) = \textstyle\sum_{i=1}^{\infty} c(\omega, i) \cdot k_i(\omega, U). \tag{4}$$

Suppose that each k_i is a Green kernel with respect to μ, with ratio $r_i : \Omega \times \Omega \to [0, \infty]$ (Def. 5.1). Suppose that we can always detect which kernel was used, i.e. there is a function $e : \Omega \times \Omega \to \mathbb{N}$ such that $\int [e(\omega, \omega') = i]\, k_i(\omega, \mathrm{d}\omega')\, \mu(\mathrm{d}\omega) = 1$.

Theorem 7.1. *The kernel k (4) is a Green kernel with respect to μ, with ratio $r \colon \Omega \times \Omega \to [0, \infty]$*

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

7.2 Single-site proposal kernel for lazy rose trees

Recall the representation of probabilistic programs developed in Sections 4 and 5, with Ω the infinite rose trees, and weight function $\ell : \Omega \to [0, \infty]$, and an outcome $\Omega \to X$. We describe the single-site proposal kernel at this level.

7.2.1 High level view. We now instantiate state-dependent mixing as follows. We work up-to a bijection between natural numbers and paths through the rose tree, which are countably infinite.

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

7.2.2 Calculating the nodes that are used in evaluation. 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). By going under the hood and inspecting system memory (from within Haskell) we are able to calculate this set of sites S_ω.

Crucially, we make use of GHC's `ghc-heap` module, which exposes the layout of heap objects in memory allowing us to view function closures, pointers to thunked objects, and several other GHC runtime-specific objects present in memory. By analysing the parts of the tree that have been evaluated to weak-head normal form and parts still untouched (present in memory as thunks), we are able to safely inspect the *runtime evaluation state* of our random tree without forcing any further computation on it. We encapsulate our use of `ghc-heap` in a method truncating lazy infinite rose trees into finite partial trees, as follows. We first define the type of partial trees where unevaluated branches are explicitly marked.

```
data PTree = PTree (Maybe Double) [Maybe PTree]
```

Following this we define `trunc :: Tree → IO PTree`, the key function we use in our implementation of single-site MH. It takes a lazy rose tree as input, inspects its structure in the Haskell heap, and from it builds the partial tree. For example, if the evaluation of a program forces the partial evaluation of a rose tree in memory to be `t = Tree _ (_ : (Tree 0.3 _): _ : (Tree 0.4 _): _)` (underscores represent unevaluated thunks), its `trunc`ation will return `PTree Nothing [Nothing, Just (PTree (Just 0.3) []), Nothing, Just (PTree (Just 0.4) [])]`. It is from this partial tree that we identify the finite set of used nodes S_ω, which is the key ingredient in the proposal step (§7.2.1). We then uniformly pick a site to modify, and follow the generic Metropolis-Hastings-Green algorithm (§5.3).

7.2.3 A note on irreducibility. A Markov chain constructed using our single-site MH need not always be irreducible: this depends on the program. This can immediately be seen in the coin-toss example `test` from §6.1 where the only way to go from `(x,y) = (True,True)` to `(False,False)` is via either `(True,False)` or `(False,True)` (since only one variable can be changed at a time), but those are `score 0` regions which will never get accepted, forcing all our samples to be constant.

To illustrate why formal irreducibility should perhaps not be an end goal in itself, notice that we can make this particular kernel formally irreducible by replacing the `score 0` with an extremely small non-zero value. But in practice our samples will still be constant due to the tiny probability of accepting the intermediate sample.

Another general approach would be to use Theorem 7.1 to mix single-site MH with another kernel that is more reliably irreducible, such as the kernel from Section 5.

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 supports a primitive memoization construct. This leads to a programming style for lazy behaviour: instead of writing `{ x ← t ; y ← u ; z ← v ; ...}` and expecting lazy evaluation, one can write

```
f ← sample (memoize \i → case i of {1 → t ; 2 → u ; 3 → v}) ; ...
```

with eager evaluation, and use `f 1`, `f 2`, `f 3` in place of `x`, `y`, `z` respectively. Although this is an unusual programming style, it is useable nonetheless. Since Church is untyped, the connection with synthetic measure theory is less clear, but the connection with non-parametric statistics is heavily emphasised, for example in the analysis of stick-breaking [24, 54] and exchangeable primitives [78]. In summary, from a bird's eye view, LazyPPL is a variation of Church with more idiomatic laziness and a type system that gives a connection with synthetic measure theory.

Languages such as Anglican [72], WebPPL [25], BayesDB [58] and Turing [6] follow within the tradition of Church, exploring ideas from non-parametric statistics further.

The Birch language [46] is possibly more practically focused than Church. It is class-based, transpiling to C++, rather than a fully functional language, and so the connection to synthetic measure theory is less clear. But Birch heavily uses laziness and advanced control flow manipulations in its inference methods [44, 45], and in this regard it is considerably more advanced than the LazyPPL implementation.

Beyond these examples, laziness has been explored in various aspects of probabilistic programming, dating back at least as far as the pioneering work by Koller et al. [39], and more recently in the work on lazy factored inference in Figaro [52], and efficient implementation in delimited continuations through Hansei ([35], 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 [47, 48, 75] is a major example, providing a DSL with impressive symbolic inference methods. Stochaskell [53] also provides a DSL, embedded in Haskell, 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 [60], which is a monad-based implementation of a variety of inference combinators, also inspired by the formalism of quasi-Borel spaces [61]. MonadBayes is not fully lazy: the Metropolis-Hastings simulation is based on the state monad and does not support laziness. The LazyPPL project grew out of adding laziness to MonadBayes, leading to the developments in this paper, and to the more natural expression of the various examples in this paper. For simplicity of exposition here we have focused here on the 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 have also been used as a foundation for a new dependent type system based on 'trace types' [40] (which has been prototyped in Haskell). This provides a potential well-typed foundation for the 'programmable inference' that makes recent languages such as Gen [14] and Pyro [4] so powerful in practice. 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, 59]) and functional languages for probabilistic network verification ([74], following [15]).

8.3 Other implementations of synthetic probability theory

Finally we note two other approaches to implementing synthetic 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 [68], inspired by categorical constructions over Markov categories [16, 69]. These approaches are currently focused on more refined notions of conditional probability that are possible in more restricted situations (respectively, discrete probability and Gaussian 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 LazyPPL: a lazy probabilistic programming library providing two monads (for probability and measure, §2) and two new Metropolis-Hastings-based algorithms (§5, §7). The methods are based on recent foundations from quasi-Borel spaces and synthetic probability theory

(§4). We have shown that the resulting forms of laziness are a useful modelling idiom for a variety of Bayesian models, including piecewise linear regression (§3.1.2), non-parametric clustering (§3.2), Wiener process regression (§6.4), and non-parametric feature extraction (§6.5).

REFERENCES

[1] N. L. Ackerman, J. Avigad, C. E. Freer, D. M. Roy, and J. M. Rute. Algorithmic barriers to representing conditional independence. In *Proc. LICS 2019*.

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

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. 18(9):509–517.

[4] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

[5] D. M. Blei, T. L. Griffiths, and M. I. Jordan. The nested chinese restaurant process and bayesian nonparametric inference of topic hierarchies. *Journal of the ACM (JACM)*, 57(2):1–30, 2010.

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

[7] J. Borgstrom, U. Dal Lago, A. D. Gordon, and M. Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *Proc. ICFP 2016*, 2016.

[8] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

[9] K. Cho and B. Jacobs. The EfProb library for probabilistic calculations. In *Proc. CALCO 2017*, 2017.

[10] K. Cho and B. Jacobs. Disintegration and Bayesian inversion via string diagrams. *Math. Struct. Comput. Sci.*, 29:938–971, 2019.

[11] K. Cho, B. Jacobs, B. Westerbaan, and A. Westerbaan. An introduction to effectus theory. arxiv:1512.05813, 2015.

[12] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. ICFP 2000*, pages 268–279, 2000.

[13] B. Coecke. Terminality implies non-signalling. In *Proc. QPL 2014*, 2014.

[14] M. F. Cusumano-Towner, F. A. Saad, A. K. Lew, and V. K. Mansinghka. Gen: a general-purpose probabilistic programming system with programmable inference. pages 221–236, 2019.

[15] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva. Probabilistic NetKAT. In *Proc. ESOP 2016*, 2016.

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

[17] T. Fritz, T. Gonda, and P. Perrone. De Finetti's theorem in categorical probability. *Journal of Stochastic Analysis*, 2(4), 2021.

[18] T. Fritz, T. Gonda, P. Perrone, and E. F. Rischel. Representable Markov categories and comparison of statistical experiments in categorical probability, 2020.

[19] C. Führmann. Varieties of effects. In *Proc. FOSSACS 2002*, 2002.

[20] C. Geyer. Introduction to Markov Chain Monte Carlo. In *Handbook of Markov Chain Monte Carlo*. Chapman Hall/CRC, 2011.

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

[22] M. Giry. A categorical approach to probability theory. *Categorical Aspects of Topology and Analysis. Lecture Notes in Mathematics*, 1982.

[23] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2003.

[24] N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. 2008.

[25] N. D. Goodman and A. Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2020-10-15.

[26] B. Gram-Hansen, C. S. de Witt, R. Zinkov, S. Naderiparizi, A. Scibior, A. Munk, F. Wood, M. Ghadiri, P. Torr, Y. W. Teh, A. G. Baydin, and T. Rainforth. Efficient Bayesian Inference for Nested Simulators. *2nd Symposium on Advances in Approximate Bayesian Inference (AABI)*, 2019.

[27] P. J. Green. Reversible jump Markov Chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82(4):711–732, 1995.

[28] T. Griffiths and Z. Ghahramani. The Indian buffet process: An introduction and review. *Journal of Machine Learning Research*, 12(32):1185–1224, 2011.

[29] C. Heunen, O. Kammar, S. Staton, and H. Yang. A convenient category for higher-order probability theory. In *Proc. LICS 2017*, 2017.

[30] R. Hinze. Memo functions, polytypically! In *Proceedings of the 2nd Workshop on Generic Programming, Ponte de*, pages 17–32, 2000.

[31] B. Jacobs. Semantics of weakening and contraction. *Ann. Pure Appl. Logic*, 69, 1994.

[32] B. Jacobs. Probabilities, distribution monads, and convex categories. *Theoret. Comput. Sci.*, 412, 2011.

[33] A. Kechris. *Classical Descriptive Set Theory.* Springer, 1987.

[34] O. Kiselyov. Problems of the lightweight implementation of probabilistic programming. In *Proc. PPS 2016*, 2016.

[35] O. Kiselyov and C. Shan. Embedded probabilistic programming. In *Proc. DSL 2009*, 2009.

[36] A. Kock. Monads on symmetric monoidal closed categories. *Arch. Math.*, 21, 1970.

[37] A. Kock. Bilinearity and cartesian closed monads. *Math. Scand.*, 29, 1971.

[38] A. Kock. Commutative monads as a theory of distributions. *Theory and Applications of Categories*, 26(4), 2012.

[39] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *Proc. AAAI 1997*, 1997.

[40] A. K. Lew, M. F. Cusumano-Towner, B. Sherman, M. Carbin, and V. K. Mansinghka. Trace types and denotational semantics for sound programmable inference in probabilistic languages. In *Proc. POPL 2020*, 2020.

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

[42] D. Michie. 'Memo' functions and machine learning. *Nature*, 218, 1968.

[43] E. Moggi. Notions of computation and monads. *Information and Computation*, 1991.

[44] L. Murray, D. Lundén, J. Kudlicka, D. Broman, and T. Schön. Delayed sampling and automatic Rao-Blackwellization of probabilistic programs. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, pages 1037–1046, 2018.

[45] L. M. Murray. Lazy object copy as a platform for population-based probabilistic programming. arxiv:2001.05293, Jan 2020.

[46] L. M. Murray and B. Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 2018.

[47] P. Narayanan, J. Carette, W. Romano, C. Shan, and R. Zinkov. Probabilistic inference by program transformation in Hakaru (system description). In *Proc. FLOPS 2016*, pages 62–79, 2016.

[48] P. Narayanan and C. Shan. Symbolic disintegration with a variety of base measures. *ACM Transactions on Programming Languages and Systems*, 42(2), 2020.

[49] D. Navarro and T. Griffiths. A nonparametric Bayesian method for inferring features from similarity judgments. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems*, volume 19. MIT Press.

[50] P. Orbanz and D. M. Roy. Bayesian models of graphs, arrays and other exchangeable random structures. *IEEE Trans. Pattern Anal. Mach. Intell.*, 37(2):437–461, 2015.

[51] P. Panangaden. *Labelled Markov Processes.* World Scientific, 2009.

[52] A. Pfeffer, B. Ruttenberg, A. Sliva, M. Howard, and G. Takata. Lazy factored inference for functional probabilistic programming. arxiv:1509.03564.

[53] D. A. Roberts, M. Gallagher, and T. Taimre. Reversible jump probabilistic programming. In K. Chaudhuri and M. Sugiyama, editors, *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, volume 89 of *Proceedings of Machine Learning Research*, pages 634–643. PMLR, 16–18 Apr 2019.

[54] D. Roy, V. Mansinghka, N. Goodman, and J. Tenenbaum. A stochastic programming perspective on nonparametric Bayes. In *Proc. Workshop on Non-Parametric Bayes*, 2008.

[55] D. M. Roy. The continuum-of-urns scheme, generalized beta and indian buffet processes, and hierarchies thereof. *arXiv preprint arXiv:1501.00208*, 2014.

[56] D. M. Roy, N. Ackerman, J. Avigad, C. Freer, and J. Rute. Exchangeable graphs, conditional independence, and computably-measurable samplers. Talk at CCA 2013. http://cca-net.de/cca2013/slides/17_Daniel%20Roy.pdf.

[57] D. M. Roy and Y. Teh. The Mondrian Process. In *Advances in Neural Information Processing Systems*, volume 21. Curran Associates, Inc.

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

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

[60] A. Ścibior, O. Kammar, and Z. Ghahramani. Functional programming for modular bayesian inference. In *Proc. ICFP 2018*, 2018.

[61] A. Ścibior, O. Kammar, M. Vákár, and S. Staton. Denotational validation of higher-order Bayesian inference. *Proceedings of POPL*, 2018.

[62] D. Shiebler. Categorical stochastic processes and likelihood. In *Proc. ACT 2020*, 2020.
[63] A. Solar-Lezama. The sketching approach to program synthesis. In *Proc. APLAS 2009*, 2009.
[64] S. Staton. Probabilistic programs as measures. In *Foundations of Probabilistic Programming*. CUP, 2020.
[65] S. Staton, D. Stein, H. Yang, L. Ackerman, C. E. Freer, and D. M. Roy. The Beta-Bernoulli process and algebraic effects. 2018.
[66] S. Staton, H. Yang, N. L. Ackerman, C. Freer, and D. Roy. Exchangeable random process and data abstraction. In *PPS 2017*, 2017.
[67] G. L. Steele Jr, D. Lea, and C. H. Flood. Fast splittable pseudorandom number generators. In *Proc. OOPSLA 2014*, 2014.
[68] D. Stein. GaussianInfer. https://github.com/damast93/GaussianInfer, 2021.
[69] D. Stein and S. Staton. Compositional semantics for probabilistic programs with exact conditioning. In *Proc. LICS 2021*, 2021.
[70] A. Stuhlmuller and N. D. Goodman. Reasoning about reasoning by nested conditioning. *Cognitive Systems Research*, 28, 2014.
[71] L. Tierney. Markov chains for exploring posterior distributions. *The Annals of Statistics*, 22(4):1701–1728, 1994.
[72] D. Tolpin, H. Yang, J. W. van de Meent, and F. Wood. Design and implementation of probabilistic programming language Anglican. 2016.
[73] J.-W. van de Meent, B. Paige, H. Yang, and F. Wood. An introduction to probabilistic programming. 2018.
[74] A. Vandenbroucke and T. Schrijvers. Plonk: functional probabilistic NetKat. In *Proc. POPL 2020*, 2020.
[75] R. Walia, P. Narayanan, J. Carette, S. Tobin-Hochstadt, and C. Shan. From high-level inference algorithms to efficient code. In *Proc. ICFP 2019*, 2019.
[76] D. Wingate, A. Stuhlmueller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proc. AISTATS 2011*, 2011.
[77] F. D. Wood, C. Archambeau, J. Gasthaus, L. James, and Y. W. Teh. A stochastic memoizer for sequence data. In *Proc. ICML 2009*, 2009.
[78] J. Wu. Reduced traces and JITing in church. Master's thesis, MIT, 2013.
[79] Y. Zhang and N. Amin. Reasoning about "reasoning about reasoning". In *Proc. POPL 2022*, 2022.