LazyPPL: laziness and types in non-parametric probabilistic programs

Hugo Paquet
Department of Computer Science
University of Oxford, UK
hugo.paquet@cs.ox.ac.uk

Sam Staton
Department of Computer Science
University of Oxford, UK
sam.staton@cs.ox.ac.uk

Abstract

We introduce LazyPPL, a prototype probabilistic programming library for Haskell. The library emphasises the clarifying power of types, and the connection between non-parametric, stochastic processes and lazy (call by need) evaluation. We illustrate the power of the language with natural specifications of infinite structures including Poisson point processes, Gaussian processes, and Dirichlet Process clustering.

1 Introduction

Probabilistic programming is a powerful method for machine learning and statistics: one defines and combines Bayesian statistical models by writing programs. This paper introduces LazyPPL, a library that advances probabilistic programming using ideas from programming languages research, primarily *laziness* and *types*.

Laziness In programming languages, 'lazy' evaluation is also known as 'call-by-need': you only run the bits of programs that are needed for the result. In probabilistic programming, this becomes 'sample-by-need': you only sample from distributions that are needed for the result. In LazyPPL, we illustrate the power of this idea for non-parametric models such as Poisson point processes, Gaussian processes, and Dirichlet Process clustering and Indian Buffet Process feature models. Generally speaking, a non-parametric model has a parameter space in which the dimension is unspecified, or regarded as potentially infinite. Bayesian inference can still be tractable, because the data and output are finite, and so not all the parameters will turn out to be relevant. The **key idea** of LazyPPL is that the evaluation engine of a standard lazy programming language (Haskell) can be trusted to work out which parameters are relevant, leaving the programmer free to write potentially infinite dimensional models.

Types A second highlight of LazyPPL is that these non-parametric constructions are first class and strongly typed (§2). This means that they can be manipulated in the same way as standard parametric constructions in a traditional probabilistic programming language, such as normal distributions and gamma distributions. The type system allows this manipulation of the constructions to happen in a principled way. It also clarifies the connection between the programming constructions and the mathematical objects that they correspond to. For example, the type system says that a Poisson point process is a distribution over possibly infinite sets of points, and that the Wiener process is a distribution over functions $\mathbb{R} \to \mathbb{R}$. The Wiener process can then be applied to arguments or composed with other arithmetic or random functions, since it is treated as any other function in a programming language. We give concrete examples in Section 3.

LazyPPL is very expressive and clear, and reasonably efficient for a prototype language. But at the same time it is easy to read through the implementation: the core library is around 50 lines of

35th Conference on Neural Information Processing Systems (aiplans 2021), Sydney, Australia.

Haskell, including a Metropolis-Hastings simulation. The full library and all examples can be found at `https://bitbucket.org/samstaton/lazyppl`.

Context and related work LazyPPL appears to be the first library combining the ideas above, but it is building on plenty of prior work in probabilistic programming. Church [5, 17] and Anglican [24] support all the statistical models in this paper, although they are not lazy or typed (Church via thunking and XRPs, Anglican via its absorb/produce idiom). In a sense, we are clarifying that earlier work via types and laziness. Later languages along these lines include BayesDB [19], WebPPL [6], and Turing, where laziness is also emphasised [2]. Several earlier probabilistic programming languages include some forms of laziness, from the early days [9] and more recently including Birch [12], Figaro [15], and Hansei [8], but this is mostly without an emphasis on Gaussian Processes etc..

Turning to types, Haskell is a popular language, and Haskell libraries Hakaru [13], MonadBayes [20] and Stochaskell [16] predate LazyPPL. They support more advanced inference, but to our knowledge, they do not support non-parametric models in the way that LazyPPL does. The emphasis on types and abstraction builds on [23, 22]. These developments have all been a big inspiration for LazyPPL.

2 Overview of the language

Types and probabilities In a typed probabilistic programming language it is clear from the type signature what kind of statistical model the program describes. So a type represents a space (such as $\mathbb{R}$, or a finite space like $\{\mathsf{true}, \mathsf{false}\}$, or a space of functions), and a probabilistic program represents a probability measure on that space.

In LazyPPL, which is a Haskell library, for every type `a` there is a type `(Prob a)`. If `a` represents a space then `(Prob a)` represents the space of probability measures on that space. For example, the normal distribution, parameterized by mean and standard deviation, has type `normal :: Double -> Double -> Prob Double`.

In the non-parametric setting, the spaces and probability distributions can be complex, and the types are elucidating. For example:

- `Double -> (Prob Double)` is the type of parametrized probability measures over $\mathbb{R}$, such as a normal distribution parameterized by a mean (with some fixed variance);
- `Prob (Double -> Double)` is the type of probability measures on the space of functions $\mathbb{R} \to \mathbb{R}$ (aka random functions), such as a random linear function or a Gaussian process;
- `Prob (Prob Double)` is the type of *random* probability measures, e.g. a Dirichlet process.
- `Prob [Double]` is the type of one-dimensional point processes, i.e. random sequences (potentially infinite), such as the Poisson point process.

As a more advanced idea, we also use *abstract* data types to hide the specific choice of internal representation for a process. This makes for cleaner modelling, and emphasises the symmetries in certain stochastic processes [23]. For instance, an interface for the Chinese Restaurant Process may define abstract types `Restaurant` and `Table`, together with two functions

- `newRestaurant :: Double -> Prob Restaurant`
- `newCustomer :: Restaurant -> Prob Table`,

encapsulating the implementation details. The concrete representation could be one of many, e.g. following a stick-breaking construction (see §3.3) or a Blackwell-MacQueen urn scheme. We have also implemented the Indian Buffet Process [7] and the Mondrian Process [18] using this style of programming, using abstract types which highlight their symmetries. More generally, LazyPPL is a promising tool for understanding data modelled as an exchangeable random structure (see e.g. [14]).

Conditioning and measures Probability distributions are one key part of probabilistic programming, and the other key part is incorporating data and observations. LazyPPL is based on the general idea of Bayesian inference via weighted Monte Carlo simulation [25, §4]. Semantically, this amounts to a program describing an *unnormalized* measure, and inference is about producing a sample from that measure. (See also [21].) To capture this in LazyPPL, we associate to every

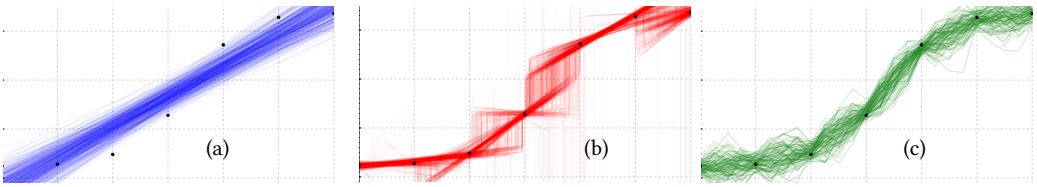

Figure 1: Bayesian regression in LazyPPL for the data set indicated by the dots. We illustrate three different priors on the function space: (a) linear, (b) piecewise linear, and (c) Wiener.

type `a`, a type `(Meas a)`, and if `a` describes some space then `(Meas a)` describes the space of possibly-unnormalized measures on that space. These are built in two ways [25, §2.1]:

- `sample :: Prob a -> Meas a` – sample from a probability measure;
- `score :: Double -> Meas ()` – update the weight of the trace (aka *factor* or *observe*), typically with a likelihood or density function, e.g. `score (normalPdf 0 1 x)`.

Inference LazyPPL provides a Metropolis-Hastings algorithm which we use to obtain samples from a distribution of type `(Meas a)`. The algorithm returns an infinite stream of samples, computed lazily as needed for plotting or analysing.

The challenge for a sampling algorithm like Metropolis-Hastings is to quickly produce samples with high weight. For LazyPPL we implemented a novel general-purpose version of M-H which works suprising well. The algorithm is parameterized by a probability p, and at each step, traverses the current program trace and re-samples each variable with probability p. This produces a new trace, which we accept or reject as usual. For a lazy language like LazyPPL this is a convenient proposal kernel because it does not require knowing the number of sample sites in the trace in advance. Our random seed comprises infinite sequences of values, and the traversal is itself lazy: the new trace may be longer or shorter, and Haskell re-samples exactly what is needed.

3 Examples of LazyPPL models

We now give examples, focusing on non-parametric models. Where LazyPPL comes into its own is that the model can involve natural types and structures that have unbounded or infinite dimension.

3.1 A simple parametric model: linear regression

We can perform linear regression in LazyPPL by first defining random linear functions:

```
linear :: Prob (Double -> Double)
linear = do a <- normal 0 3
            b <- normal 0 3
            let f x = a*x + b
            return f
```

In general, we perform Bayesian regression by sampling from some prior distribution on the function space, and recording a weight for each data point, with the weight coming from some Gaussian noise.

```
regress :: Double -> Prob (a -> Double) -> [(a,Double)] -> Meas (a -> Double)
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 1(a).

3.2 Non-parametric regression

The function `regress` can be used for more involved kinds of regression. For a first *non*-parametric model, we consider *piecewise* linear regression, where the prior is over piecewise linear functions

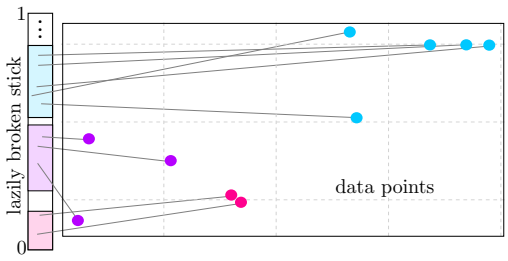

Figure 2: Dirichlet process clustering by stick-breaking in LazyPPL.

with an infinite number of pieces. We define a function that will splice together different draws from a random function, given a *point process* of change points:

```
splice :: Prob [Double] -> Prob (Double -> Double) -> Prob (Double -> Double)
```

We can define a *Poisson* point process with given rate, by sampling steps from exponential distributions:

```
poissonPP :: Double -> Prob [Double]
```

Then, piecewise linear regression, as shown in Figure 1(b), can be implemented as `regress 0.1 (splice (poissonPP 0.1) linear) dataset`. One key point of laziness is that the computation is finite even when distributions are clearly over infinite-dimensional structures.

LazyPPL also supports regression with Gaussian processes. For simple presentation, we stick to the Wiener process, also known as Brownian motion. In LazyPPL we have a random function

```
wiener :: Prob (Double -> Double)
```

that can be used to perform Wiener process regression, `regress 0.1 wiener dataset`, as shown in Figure 1(c). This is also infinite dimensional, but still returns results because the inference algorithm is lazy. It is a first class random function, so we can straightforwardly transform it with non-random functions, or combine it with other random functions such as `splice`.

3.3 Dirichlet process clustering and stochastic memoization

Dirichlet process clustering allows for an unknown number of clusters. One implementation is via 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 in LazyPPL:

```
stickBreaking :: Double -> Double -> Prob [Double]
stickBreaking alpha lower =
  do r <- beta 1 alpha
     let v = r * (1 - lower)
     vs <- stickBreaking alpha v
     return (v : vs)
```

It is now easy to define the Dirichlet process:

```
dp :: Double -> Prob a -> Prob (Prob a)
```

As the type suggests, `(dp alpha p)` provides a *random* distribution over clusters. It does this by first performing stick-breaking and then assigning a draw from `p` to every cluster. We use this to perform Bayesian clustering of a data set, using code similar to `regress` above (Fig. 2).

Memoization is a form of laziness where a function caches previous results instead of recalculating [11]. In a stochastic setting, memoization has been proposed as a powerful way of building infinite-dimensional probability measures (e.g. [5, 17, 26]). In LazyPPL, we have:

```
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`). In the clustering example, we use memoization to assign facets to each cluster.

Feature extraction is a generalization of clustering, where each point can belong to more than one feature. For example, a movie belongs to multiple genres. We have used laziness in LazyPPL to implement the Indian Buffet Process [7], which supports an unbounded number of features.

4 Summary and discussion

LazyPPL is a prototype Haskell library that supports types and lazy probabilistic programming. The truncations of Bayesian non-parametrics are outsourced to the general purpose Haskell implementation of laziness, both in the models and in the Metropolis-Hastings inference. We illustrated this with non-parametric regression (§3.2) and clustering (§3.3).

Probabilistic programming makes it easy to build models in a modular way, and the type system of LazyPPL emphasises this aspect. For example, it is very easy to switch the Poisson process for a different point process in the piecewise regression example (Fig. 1). We can also compose models to explore hierarchical versions of the Dirichlet process, Gaussian processes and so on.

LazyPPL opens up some further research challenges. On the practical side, although our novel MH algorithm works well for simple examples, there is more work to combine typed lazy programs with the fast implementations that are used for specific non-parametric models (e.g. [10]). We have not included performance graphs here because the main aim here is a proof of concept of the expressiveness of the language. Going forward, we may follow in the substantial progress made recently for parameteric models (e.g. [1, 3]).

On the side of programming language theory, there is a challenge to identify a class of primitives that are useful for programming in the style of LazyPPL. Although most of our models are vanilla Haskell, we occasionally use caches to implement memo tables; this is encapsulated carefully. Despite the caching and laziness, we can reorder lines of a LazyPPL program. This is a programmer's statement of Fubini's theorem, which holds because the processes we consider are exchangeable (e.g. [23]). We argue that this allows us to view LazyPPL as a synthetic probability theory [4].

Acknowledgements. We acknowledge funding from a Royal Society University Research Fellowship, the ERC BLAST grant, and the Air Force Office of Scientific Research under award number FA9550-21-1-0038. We are grateful to Nate Ackerman, Cameron Freer, Ohad Kammar, Alex Lew, Dan Roy, Adam Ścibior, Ken Shan, Hongseok Yang and the Oxford group for many helpful discussions about topics in this paper.

References

[1] 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.

[2] 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.

[3] 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.

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

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

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

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

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

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

[10] A. R. Kosiorek, H. Kim, I. Posner, and Y. W. Teh. Sequential attend, infer, repeat: Generative modelling of moving objects. In *Proc. NeurIPS 2018*, 2018.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] D. M. Roy and Y. Teh. The mondrian process. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems*, volume 21. Curran Associates, Inc., 2009.

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

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

[21] S. Staton. Probabilistic programs as measures. In *Foundations of Probabilistic Programming*. CUP, 2020.

[22] S. Staton, D. Stein, H. Yang, L. Ackerman, C. E. Freer, and D. M. Roy. The Beta-Bernoulli process and algebraic effects. 2018.

[23] S. Staton, H. Yang, N. L. Ackerman, C. Freer, and D. Roy. Exchangeable random process and data abstraction. In *PPS 2017*, 2017.

[24] D. Tolpin, H. Yang, J. W. van de Meent, and F. Wood. Design and implementation of probabilistic programming language Anglican. 2016.

[25] J.-W. van de Meent, B. Paige, H. Yang, and F. Wood. An introduction to probabilistic programming. 2018.

[26] F. D. Wood, C. Archambeau, J. Gasthaus, L. James, and Y. W. Teh. A stochastic memoizer for sequence data. In *Proc. ICML 2009*, 2009.