# School of Informatics

## Informatics Research Review
## Probabilistic Programming

**Oliver Goldstein (s1424164)**
**January 2019**

**Abstract**

I'll look at a representative selection of probabilistic programming languages, giving a brief introduction to the theoretical goals of probabilistic programming. I choose the representatives based on the main principles of probabilistic programming in each language, i.e. random variables, conditioning and inference. For each language and for each principle, I detail the approach taken. I conclude by addressing nuances of comparison in the field and proposing potential future work.

Signature: [illegible]

Date: 13th January 2019

**Supervisor:** Björn Franke

## Contents

## 1 Introduction

This review will cover different types of *probabilistic programming languages*. In 2018, de Meent et al. [1] argue that the recent successes of artificial intelligence have been caused by the emergence of programming language tools, such as NumPy, that are able to automatically engage in the ‘tedious’ derivation and calculation of gradients. Probabilistic programming languages build on this idea to provide a generic way to write arbitrary models and then calculate the uncertainty of parameter choices.

Languages that are said to be probabilistic are defined as those that support both random sampling from distributions and the conditioning of values of a variable via observations [2]. The program calculates the posterior provided a model and an inference strategy, where inference is broadly defined by Gordon et al. [2] as the problem of computing an explicit probability distribution implicit in the program. The idea that a program can represent a probabilistic model goes back to 1978 [3] and programming languages for Bayesian modelling have existed since at least 1994 [4]. These languages can be used for a wide variety of inference tasks such as those listed at [5][6], which mentions problems as varied as reasoning about reasoning, belief propagation and air conditioning calibration, through to the widely cited TrueSkill Engine that Xbox uses to match players such that there is a 50/50 probability of either player winning the match. To emphasise the importance of these paradigms, it is noteworthy that probabilistic languages are developed by high profile corporations such as Microsoft (Infer.NET) [7], Uber (Pyro) [8] also Google (Edward) [9].

This document is structured as follows. Section 2 introduces the background knowledge needed to understand basic inference. Section 3 introduces the languages that have been explored extensively and the main features of probabilistic programming, random variables, the ability to condition on new data and different methods of inference. Section 4 concludes with a simple evaluation of the research area.

## 2 Background

Probabilistic programs implicitly specify a probability distribution, be that over logic programs, often termed worlds, or over data. **Inference**, is the task of given data, to recover a probability distribution over parameters that generated that data. However, the definition of inference changes in subtle ways.

Probabilistic models can be specified visually by factor graphs, which are bipartite graphs that express function factorisation. Many forms of inference are created with specific factor graphs in mind, such as Hidden Markov Models or Markov Chains, see Figure 1. Probabilistic programming languages serve as a tool to codify these graphs [10]. The goal of probabilistic programming as mentioned by [2] is to hide inference inside the compiler to allow the programmer to focus on writing models and have inference automatically done for them. However, an important negative result [11] demonstrates that this automation can not be done in the continuous case, because the computation of conditional probability distributions is sometimes impossible as they reduce to the halting problem [12], and where automation can be done, the process may be inefficient [13]. Probabilistic programming offers users two important benefits. First, as models become more complex, the formalism for describing them starts to matter. For example, graphical models scale awkwardly visually. Second, probabilistic programming can help encapsulate the often messy details of inference from the statistical model it analyses.

In doing inference, the goal is to obtain the **posterior**. The posterior, $P(\theta|D)$ measures uncertainty over parameters, where $\theta$ are the parameters of the model and $D$ are observed data. The posterior is rooted in Bayesian philosophy [15] as model parameters are ascribed an associated uncertainty. An example model parameter is the bias of a coin and data could be coin tosses already seen.

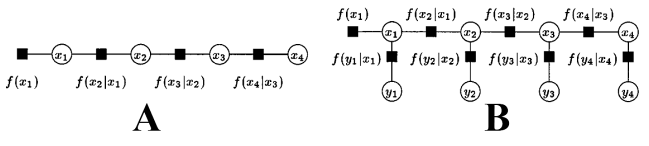

Figure 1: Factor graph of Markov Chain (A) and HMM (B): source [14]

In Bayesian inference, the model includes a prior distribution $P(\theta)$ over parameters. In order to calculate a probability distribution over the parameters, the joint probability $P(D|\theta)P(\theta)$ is calculated and subsequently normalised by the model evidence $P(D)$, which is just Bayes rule. The joint probability is a random set of examples drawn from the **prior** distribution, multiplied by the **likelihood** distribution function, which expands via the product rule when given multiple data.

Normalisation of Bayes Rule is not needed in a maximum likelihood estimate, which will find a theta, that maximises the probability of the data, subject to the model at hand. If a 'good' $\theta$ value is found, the parameters of the system are often updated and a new posterior is calculated. Normalisation is hard because of the many terms involved in the calculation of the model evidence. The model evidence expands via the law of total probability, in both the continuous and discrete cases, revealing in the discrete case, a potentially intractable number of terms, which increases exponentially with the dimensionality of the prior over the parameters, and likewise, in the continuous case a potentially intractable integral. Calculation of the marginal likelihood, and thus the posterior, is made easier if the prior is conjugate (or has the same form) to the posterior, because then there is often an algebraic closed form solution and can result in exact inference.

*Exact inference* refers to calculating the posterior exactly, which means computing every term in the discrete case or the integral in the continuous case. Often, evaluation of the likelihood function is computationally inefficient, and so approximation methods like variational inference [16] are used. These model the likelihood with a proxy function before minimising measures, typically the KL-divergence, to closely mimic the original underlying distribution. These divergence measures can be estimated with techniques such as Stochastic Variational Inference [17] which uses gradient descent to give estimates of the value of the divergence between approximate and posterior distributions. In addition to variational and exact inference, Monte Carlo (MC) methods are also used. These sample from the joint distribution, building up the posterior over time. Examples are Hamiltonian Monte Carlo [18] (HMC), Markov Chain Monte Carlo [19] (MCMC) and Sequential Monte Carlo [20] (SMC). Exact, variational and MC inference strategies are expanded on in the relevant sections.

The following short excerpt of WebPPL code, adapted from Dan Roy [13], emphasises and clarifies the essential operations that make up probabilistic programming.

```
 ...
 // Inferring the bias of a coin using rejection sampling.
var heads = true, tails = false;
var observed_data = [heads, tails, heads, heads]

 var model = function(){
   var prior = sample(Uniform({a: 0, b: 1}))
   var likelihood = function(p) {return bernoulli(p);};
   var samples = mapData({data: [prior, prior, prior, prior]}, likelihood);
   condition(samples.toString() === observed_data.toString())
   return prior
 }
 // Plot the result of inference.
 viz(Infer({method: 'rejection', samples:1000}, model))
```

The prior is sampled from the uniform interval $[0, 1]$ and the likelihood function is Bernoulli, assuming values in the set $\{0, 1\}$. The conditioning process is explicit with condition. Conditioning is the fundamental method in which statistical inference over parameters is done [13]. Only program traces that fulfill the condition are considered by the infer method, where a trace, defined by [13], is simply the tree data structure that captures the random choices encountered and the path taken by the interpreter while evaluating the program. The random primitives are in this case, real valued random variables and boolean random variables (mappings from events in the "world" to $\mathbb{R}$ or $\mathbb{B}$). The method of inference used here is a non-markov Monte Carlo approach, termed rejection sampling. Intuitively, this can be understood by considering the traces of the program that progress pass the condition statement, which will generally be dominated by prior values that tend to yield the observed data. Instead of observed data, one could also include knowledge about scientific laws relating to the world, resulting in theta that are most 'realistic'.

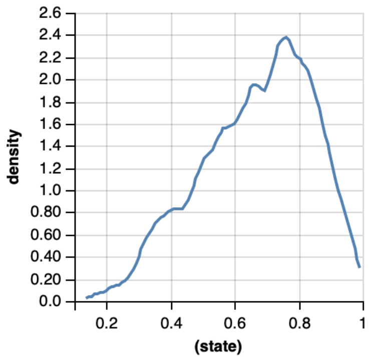

Figure 2: WebPPL inference graph of biased coin inference, based on the code. The most likely bias parameter is 0.75, which makes sense as 3/4 of the coins are heads (true).

# 3 Probabilistic Programming

## 3.1 Languages explored

My selection criteria are the alignment of the languages with respect to the historic ontologies classifying orginary programming languages. For instance, the chapters of 'Concepts of programming languages' [21] detail the main paradigms, such as functional, logical, imperative or object-oriented. These languages differ from each other and are prototypical to one of the core paradigms. This choice focusses this review, yet keeps it diverse. Unless made explicit, a statement in this review refers to the languages mentioned in Table 1. A type system can have various types of type such as behavioural types, descriptive type etc. For simplicity the types have been categorised as untyped, limited or expressive. The use of "ProbLog", unless explicitly made clear refers to ProbLog 2.

| **Language** | **Language / Library** | **Paradigm** | **Types** | **E** | **V** | **M** | **#** |
|---|---|---|---|---|---|---|---|
| WebPPL | JS / node.js | Imperative / OOP | Limited | ✓ | ✓ | ✓ | 25 |
| Edward | Python / Tensorflow | Meta / OOP | Limited | ✓ | ✓ | ✓ | 15 |
| Tabular | Infer.NET / Excel | Database-schema | Expressive |  | ✓ | ✓ | 7 |
| ProbLog 1, 2 | ProLog | Logical | Untyped | ✓ | ✓ | ✓ | 2 |
| Pyro | Python / PyTorch | OOP | Untyped | ✓ | ✓ | ✓ | 27 |
| Monad Bayes (M-B) | Haskell | Functional | Expressive |  |  | ✓ | 12 |

Table 1: Languages studied in-depth in this review.
**Legend: E** : Exact inference, **V** : Variational inference, **M** : Monte Carlo
# : Number of distribution types supported.

## 3.2 Random primitives

The random variables that are part of the language can take either continuous (arbitrary precision) or discrete form and can massively vary in the flexibility of their expression and format. One can sample from random variables, which themselves take the form of distributions and thus the support and extensibility for distribution types determines, to a large extent, the utility of the language in modelling different phenomena. Edward, Pyro, M-B, webPPL and ProbLog have an explicit sampling function, whilst Tabular implicitly samples concrete elements in the database. Edward and Pyro have very similar distributions, as PyTorch distributions were explicitly designed from the TensorFlow distributions package. All languages except for ProbLog 1 [22] support sampling from continuous and discrete distributions. ProbLog 1 only supports discrete Bernoulli random variables, which stands in contrast to Stan [23], which only supports continuous random variables. For example, webPPL [24] supports sampling from an extensive list of distribution types, such as, Bernoulli, RandomInteger and Gaussian. This is in contrast to ProbLog 1 [22], which exclusively has statements of the form 0.8 : $likes(X, Y) : -friendof(X, Z), likes(Z, Y)$, which means if friendof and likes are true, then likes is true with probability 0.8 and false with probability 0.2. These statements are only predicated on the prefix real value which parameterises a Bernoulli random variable. Guttmann et al. [25] extends ProbLog to include continuous, in particular Gaussian distributions [25], though most work in the literature only includes discrete random variables. Tabular [10] uses an alternative format to represent random variables. In particular, random variables are represented as latent variable columns in a relational schema, spreadsheet or database. In

Edward, random variables are objects parameterised by tensors. Edward builds in support for composing random variables and allows them to be inserted into the control flow of the TensorFlow [26] graph structure. This allows for inference over models that naturally express themselves as the composition of random variables such as directed graphical models [27].

Whilst all languages are theoretically extensible, they are not similar in their ease of extensibility. Edward and M-B are the only languages that make the method to extend the distributions clear and explicit in their white-papers, both requiring two functions, score and sample, to make a new one.

## 3.3 Conditioning

The condition operator ranges in its implementation. Tabular has no explicit condition operation, instead implicitly conditioning on concrete input data. Edward as an intermediary doesn't provide an explicit condition operator, but allows for conditional inference by providing training data to the $Inference$ method of the class *Ed*. On the other hand, webPPL and Pyro explicitly make use of at least two different conditioning operations. For example, both have $\mathit{condition}(\mathit{boolean})$, which only allows the execution of program traces that fulfill the condition, and $\mathit{observe}(\mathit{distribution}, \mathit{value})$ which conditions directly from a distribution, making it clear to the programmer which latent variables have inference performed over them. WebPPL, additionally includes $factor(\mathit{condition}, v1, v2)$ which adds the relevant value $v1$ or $v2$, dependent on the condition, to the log likelihood of the current trace. This is similar to M-B which uses weights to condition execution probabilities. In some cases, the conditioning process is only approximate because if one insists conditioning be exact, an intractable calculation often results, in the sense that the model may never **exactly** yield the observed data. This is intuitive when considering noisy data structures such as latent geometry that produces the generation of scenes, as each pixel would have to be exactly the same shade. As such, PICTURE [28], a DSL for scene perception, uses tolerance variables that introduce structured noise into the rendering process, making the calculations tractable by allowing some mismatch between the true and hypothesized scene. ProbLog conditions in a slightly different way, using the $\mathit{evidence}(\mathit{Literal})$ or $\mathit{evidence}(\mathit{Literal}, \mathit{Bool})$ clause, where the Literal is a grounded atomic fact about the world and the Bool parameter determines whether the evidence is a true or false fact about the world.

## 3.4 Methods of inference

Inference is defined differently dependent on the philosophy employed. For example, ProbLog 1 defines inference as computing the success probability of queries without supplied evidence. ProbLog 1 manages to do inference with up to 100,000 conjuncts, using approximation algorithms, over the NP hard problem that inference over first order logic formulas in this context represents. Whilst inference can be done exactly with a smaller number of conjuncts in ProbLog 1, there would be no improvement over its predecessor Probabilistic Datalog (pD) [29], which can at most, handle about ten conjuncts. ProbLog 2 builds on ProbLog 1, providing the ability to declare evidence, using the evidence clause, over arbitrary atoms, to define inference, through the use of *query*, in a logical paradigm that is defined as computing the conditional probability of a query given data. It is this definition of inference, that is parallel with a frequentist interpretation of probability theory. The bayesian interpretation sees inference as computing the probability of the parameters, rather than the success of a query. This divide in ideology, extends to the papers and those that are referenced, forming two distinct 'bubbles' in the literature [30].

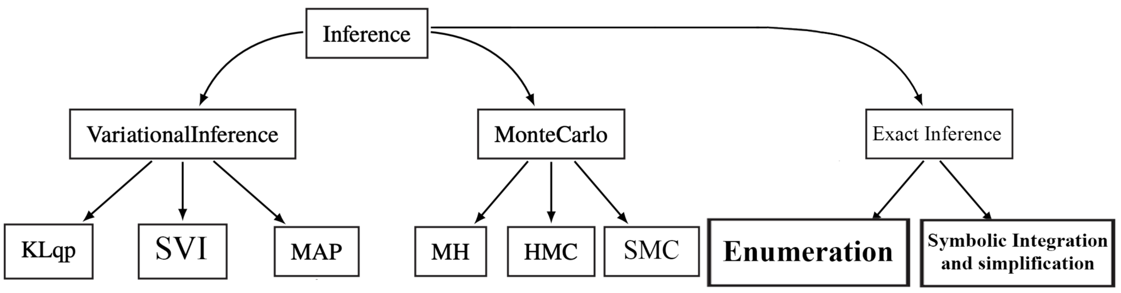

Figure 4: A visual ontology of the main classes of inference in the papers reviewed and prominent methods within those classes.

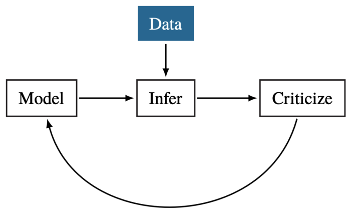

Figure 3: Box's iterative methodology, source: [9]

The languages vary in support given to inference types and the context in which inference is done. Figure 4 details the main classes of Bayesian inference. WebPPL provides inference on the web, and makes a point of being expressive, in that computation of any computable distribution can occur in any context, through explicit support for recursion and higher order functions [12]. Uber's Pyro, whilst as flexible, makes a point of providing inference for large data sets and GPUs, which makes sense if one considers their likely use in self driving cars. Pyro distinguishes itself from Edward, which also is designed for big data sets, in three ways. Firstly, Pyro allows for dynamic control flow, whereas Edward is restricted by compilation to a static TensorFlow graph. Secondly, Edward provides a framework for iterative development of models using George Edward Box's methodology, see Figure 3. As such, Edward includes data fit error metrics to help model builders assess the quality of the model (which are seen as inevitably wrong), after inference. Finally, Pyro innovates and uses a library of algebraic effect handlers called Poutine, that creates a core of composable abstractions to allow different inference algorithms to be reasoned about. Whilst they have different contexts, Pyro, WebPPL and Edward all provide inference as a method in the context of object oriented programming. Tabular, whilst just as flexible, places emphasis on a more restricted vision, providing inference in only two contexts. These two forms are query-by-latent-column and query-by-missing-values. In the former, both inputs and outputs to a system are observed and the goal is to predict the parameter values that are likely to exist for the latent columns, which are implicitly sampled from a particular distribution. In the latter, the goal is to predict the output columns, which are observable. The alternative axis on which these languages vary is the support for different inference algorithms, which is detailed in Table 1. M-B, the most recent library examined, provides an implementation of a language that closely resembles a denotational semantics of probabilistic programs. It then places emphasis on composability using monads, which instantiate themselves in the form of inference representations.

### 3.4.1 Variational Inference

WebPPL, Pyro, and Stan provide support for variational inference through the use of guide functions, that also implicitly specify a reference distribution. These distributions, can then be optimised to approximate the original, target distribution. The target program is the original model with the observe statements factored in. These guide functions generally contain a model from a simpler class of distributions. Given a similarity metric, WebPPL has an optimise function that takes the guide and target, and automatically computes the parameters of the guide function that most closely resemble the target program. Pyro is more flexible than WebPPL and Edward, offering the use of guide functions not only for stochastic variational inference but also for diverse use cases such as proposal distributions for importance sampling and MCMC. Edward doesn't use explicit guide functions, but provides variational inference as a method. One can simply provide the target and guide distributions with training data and the parameters are automatically propagated through to the underlying tensors. In the spirit of hiding as much of the inference as possible from the programmer, Stan offers ADVI [31], a form of black box variational inference which automatically synthesizes a suitable guide. Deep Stan [32] then extends this to provide guide parameters, that allows one to interface with ADVI to explicitly request particular guide distributions. Tabular, through its interface with INFER.NET, supports variational message passing [33] and expectation propagation, two forms of deterministic variational inference. Interestingly, INFER.NET is the only paradigm to implement variational message passing, which makes sense as Christopher Bishop, one of the authors of the method, works at Microsoft. ProbLog provides approximate inference algorithms for the computation of the success of a query, which is analagous to the variational approach, through the use of a k-optimal approach [34] when retrieving, evaluating and summing the exponentially many DNFs involved.

### 3.4.2 Monte Carlo Inference

Monte Carlo algorithms, such as rejection sampling, refer to sampling based algorithms. Markov Chain Monte Carlo algorithms are a special class of MC algorithms that sample from the stationary distribution of parameter probabilities and are preferred over just Monte Carlo algorithms, as they are generally more efficient. MCMC algorithms move randomness out of the model to a parameter, allowing it to be controlled. This randomness parameter, measured by entropy, is then perturbed, such that values yielding accepting program traces are optimised for. This is essentially a random walk through the space of program traces. The method in which the entropy is perturbed, and thus, how the space of traces is explored, is determined by a kernel, such as Metropolis-Hastings (MH), which includes a way to reject perturbations likely to yield rejecting traces. ProbLog is unique in that it allows for Monte Carlo inference, in a frequentist context, that of estimating frequentist confidence intervals. WebPPL and Pyro, each have two kernels for MCMC, having HMC in common. Edward has six kernels for Monte Carlo inference, more than any other language here. Tabular only uses Gibbs sampling, a special case of MH. In one paper [35], an external library, Filzbach, provides MCMC sampling for Tabular. Some languages allow adding new kernels. With Pyro and Edward, one simply has to instantiate a member of the Trace class. WebPPL includes a unique method of sampling based inference, that no other language in this review includes. In particular, WebPPL uses uses drift kernels, which allow for sampling based on previous random choices, which are useful if the choice of prior is poor. The major idea underlying M-B, are inference transformers, such as Seq and Trace, which take inference representations - basic building blocks of inference, which themselves consist of MonadInfer, that is a combination of MonadSample and MonadCond. These transformers can be put together to form algorithms like SMC, where Seq denotes the activity of doing sequential inference and Trace is used for trace based MCMC and holds a data structure with the trace and the program with modified trace. The emphasis with M-B is on modularity and flexibility to support any MCMC algorithm that uses sequential structure or the program trace. Currently, MC based gradient based approaches, like HMC, are not supported, but work on differentiation in the context of functional programming may be useful here [36], [?].

### 3.4.3 Exact Inference

Exact inference, whilst often intractable, is handled in a variety of ways. WebPPL and Pyro use a method called enumeration, which calculates the marginal likelihood by calculating all possible program traces, and then for each program trace calculating its probability. Summing the probabilities from all program traces produces the marginal likelihood which normalises subsequent probability values. Edward uses TensorFlow's computational graph to uncover conjugacy relationships between random variables, to allow computation of model evidence.

## 4 Evaluation and concluding remarks

A few points of interest, where the languages mentioned diverge in their conception, are highlighted here with intention to pave the way for future work.

### 4.1 Correctness

Four of the languages explicitly link their construction to a notion of semantics. M-B is based on a denotational semantics [37] and is implemented in Haskell, whose type signatures allow the code to be verified more easily. WebPPL has been proved to be able to express any computable probability distribution [12] and ProbLog is based off of Sato's distributional semantics [38]. Tabular and M-B, are unique in providing a rigorous calculus for the effects, expressions and typing judgements of the language. Pyro and Edward, mention no language-specific semantics to provide a grounded meaning.

### 4.2 Comparison

The literature has no agreed standard for comparing probabilistic programming systems, which presents issues when a new system is compared to existing systems. I notice the following methods. M-B compares papers qualitatively based on LoC's, and quantitatively based on execution times of inference algorithms between Anglican and WebPPL. However, these execution times may differ with the extent to which additional cores are utilised. For instance, Edward uses TensorFlow to parallelise code, WebPPL provides a threading system, yet no mention of parallelisation occurs with M-B, which casts doubt over the meaning of such a comparison. ProbLog only compares itself to pD with respect to the number of conjuncts it can handle. Pyro tests the frame update time for variational auto-encoders, an inference algorithm, against PyTorch, a non probabilistic library. Edward, doesn't even do experiments of any kind. WebPPL provides no literature on experiments against other paradigms. Tabular compares itself on a very specific model to INFER.NET. Whether there is an inherent difficulty to comparing languages or whether the field is simply fractured is unclear, though, a priori, it seems difficult to find a robust, universal set of criteria of comparison.

### 4.3 Further work

Important further work would put forth well-reasoned arguments in favour of a comparison standard, in the probabilistic programming community. This could then be factored into the review cycle, benefitting the probabilistic programming community as a whole.

## References

[1] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018.

[2] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.

[3] Nasser Saheb-Djahromi. Probabilistic lcf. In *International Symposium on Mathematical Foundations of Computer Science*, pages 442–451. Springer, 1978.

[4] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. A language and program for complex bayesian modelling. *The Statistician*, pages 169–177, 1994.

[5] GitHub. A repository for generative models., 2018. [Online; accessed 21-December-2018].

[6] T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. Papers using infer.net, 2018. [Online; accessed 22-December-2018].

[7] T. Minka, J.M. Winn, J.P. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. /Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

[8] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. Pyro: Deep universal probabilistic programming. *arXiv preprint arXiv:1810.09538*, 2018.

[9] Dustin Tran, Alp Kucukelbir, Adji B Dieng, Maja Rudolph, Dawen Liang, and David M Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

[10] Andrew D Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgstrom, and John Guiver. Tabular: a schema-driven probabilistic programming language. In *ACM SIGPLAN Notices*, volume 49, pages 321–334. ACM, 2014.

[11] Cameron E Freer and Daniel M Roy. Posterior distributions are computable from predictive distributions. In *AISTATS*, pages 233–240, 2010.

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

[13] Daniel Roy. A personal viewpoint on probabilistic programming https://simons.berkeley.edu/sites/default/files/docs/5675/talkprintversion.pdf, 2018. [Online; accessed 27-December-2018].

[14] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on information theory*, 47(2):498–519, 2001.

[15] Jeremy Orloff and Jonathan Bloom. Comparison of frequentist and bayesian inference., 2018. [Online; accessed 21-December-2018].

[16] Michael I Jordan, Zoubin Ghahramani, Tommi S Jaakkola, and Lawrence K Saul. An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233, 1999.

[17] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[18] Radford M Neal et al. Mcmc using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2(11):2, 2011.

[19] Radford M Neal. Probabilistic inference using markov chain monte carlo methods. 1993.

[20] Arnaud Doucet, Nando De Freitas, and Neil Gordon. An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer, 2001.

[21] Robert W Sebesta. *Concepts of programming languages.* 2016.

[22] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. 2007.

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

[24] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2018-12-28.

[25] Bernd Gutmann, Manfred Jaeger, and Luc De Raedt. Extending problog with continuous distributions. In *International Conference on Inductive Logic Programming*, pages 76–91. Springer, 2010.

[26] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[27] Daphne Koller, Nir Friedman, and Francis Bach. *Probabilistic graphical models: principles and techniques.* MIT press, 2009.

[28] Tejas D Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Mansinghka. Picture: A probabilistic programming language for scene perception. In *Proceedings of the ieee conference on computer vision and pattern recognition*, pages 4390–4399, 2015.

[29] Norbert Fuhr. Probabilistic dataloga logic for powerful retrieval methods. In *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 282–290. ACM, 1995.

[30] Luc De Raedt and Angelika Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015.

[31] Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David Blei. Automatic variational inference in stan. In *Advances in neural information processing systems*, pages 568–576, 2015.

[32] Javier Burroni, Guillaume Baudart, Louis Mandel, Martin Hirzel, and Avraham Shinnar. Extending stan for deep probabilistic programming. *arXiv preprint arXiv:1810.00873*, 2018.

[33] John Winn and Christopher M Bishop. Variational message passing. *Journal of Machine Learning Research*, 6(Apr):661–694, 2005.

[34] Joris Renkens, Guy Van den Broeck, and Siegfried Nijssen. k-optimal: A novel approximate inference algorithm for problog. *Machine learning*, 89(3):215–231, 2012.

[35] Sooraj Bhat, Johannes Borgström, Andrew D Gordon, and Claudio Russo. Deriving probability density functions from probabilistic functional programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 508–522. Springer, 2013.

[36] Conal M Elliott. Beautiful differentiation. In *ACM Sigplan Notices*, volume 44, pages 191–202. ACM, 2009.

[37] Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K Moss, Chris Heunen, and Zoubin Ghahramani. Denotational validation of higher-order bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(POPL):60, 2017.

[38] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, pages 715–729, 1995.