Functional Programming for Modular Bayesian Inference

Functional Programming for Modular Bayesian Inference

ADAM ŚCIBIOR, University of Cambridge, UK and MPI for Intelligent Systems, Germany
OHAD KAMMAR, University of Oxford, UK
ZOUBIN GHAHRAMANI, University of Cambridge, UK and Uber AI Labs, USA

We present an architectural design of a library for Bayesian modelling and inference in modern functional programming languages. The novel aspect of our approach are modular implementations of existing state-of-the-art inference algorithms. Our design relies on three inherently functional features: higher-order functions, inductive data-types, and support for either type-classes or an expressive module system. We provide a performant Haskell implementation of this architecture, demonstrating that high-level and modular probabilistic programming can be added as a library in sufficiently expressive languages.

We review the core abstractions in this architecture: inference representations, inference transformations, and inference representation transformers. We then implement concrete instances of these abstractions, counterparts to particle filters and Metropolis-Hastings samplers, which form the basic building blocks of our library. By composing these building blocks we obtain state-of-the-art inference algorithms: Resample-Move Sequential Monte Carlo, Particle Marginal Metropolis-Hastings, and Sequential Monte Carlo Squared. We evaluate our implementation against existing probabilistic programming systems and find it is already competitively performant, although we conjecture that existing functional programming optimisation techniques could reduce the overhead associated with the abstractions we use. We show that our modular design enables deterministic testing of inherently stochastic Monte Carlo algorithms. Finally, we demonstrate using OCaml that an expressive module system can also implement our design.

CCS Concepts: • **Mathematics of computing → Bayesian nonparametric models**; **Metropolis-Hastings algorithm**; **Sequential Monte Carlo methods**; **Resampling methods**; • **Software and its engineering → Functional languages**; **Compilers**; *Software libraries and repositories*;

Additional Key Words and Phrases: probabilistic programming, functional programming, Bayesian inference, higher-order functions, inductive types, type-classes, module systems, monads, monad transformers, machine learning, Anglican, WebPPL, Markov Chain Monte Carlo, Sequential Monte Carlo, Monte Carlo samplers

ACM Reference Format:
Adam Ścibior, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional Programming for Modular Bayesian Inference. *Proc. ACM Program. Lang.* 2, ICFP, Article 83 (September 2018), 29 pages. https://doi.org/10.1145/3236778

1 INTRODUCTION

Probabilistic programs, as used for statistical machine learning and data science, are computer programs with two specific computational effects: one for drawing random variables from probability distributions and one adjusting the relative weight of specific values of a random variable. We

Authors' addresses: Adam Ścibior, ams240@cam.ac.uk, Department of Engineering, University of Cambridge, Trumpington Street, Cambridge, CB2 1PZ, UK , Empirical Inference Deparment, MPI for Intelligent Systems, Spemannstrasse 34, Tübingen, 72076, Germany; Ohad Kammar, ohad.kammar@cs.ox.ac.uk, University of Oxford, Department of Computer Science, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK; Zoubin Ghahramani, zoubin@eng.cam.ac.uk, Department of Engineering, University of Cambridge, Trumpington Street, Cambridge, CB2 1PZ, UK , Uber AI Labs, San Francisco, USA.

2475-1421/2018/9-ART83
https://doi.org/10.1145/3236778

explain these two effects using Haskell types and monads, following Staton [2017]. For a traditional explanation of Bayesian modelling and inference we refer the reader to one of the many excellent textbooks available [Barber 2012; Bishop 2006; MacKay 2003].

A monad `m` supporting these two constructs should support the following two functions

```
random :: m Double      score :: Log Double → m ()
```

As an aside, values of type `Log Double` are non-negative reals represented internally as a logarithm. The use of logarithms replaces multiplication with addition of double-precision floating point numbers, somewhat reducing underflow issues. Each computation of type `m a` represents a *distribution* over the values of type `a`. The function `random` draws a random variable uniformly from the unit interval $[0, 1]$. The computation `score r` *scales* the distribution by the factor $r \in [0, \infty)$, and its effect is more subtle. If we ignore the scores in a given computation `c` of type `m a`, then `c` represents a *probability* distribution over values of type `a`, called the *prior* distribution. Given a value `x` of type `a`, let the *likelihood* of `x`, be the "sum" w(`x`), over all the program traces that result in `x`, aggregating the product of scores along each trace. As an example, consider the following program, often used to explain Bayesian modelling. Using the distribution `bernoulli p = fmap (< p) random`, i.e., `True` with probability `p` and `False` with probability `1-p`, define the following computation `c`:

```
rain      ← bernoulli 0.2
sprinkler ← bernoulli 0.1
let prob_lawn_wet = case (rain, sprinkler) of
      (True , True )→ 0.99
      (True , False)→ 0.70
      (False, True )→ 0.90
      (False, False)→ 0.01
score prob_lawn_wet --observe lawn wet
return rain
```

Such programs are called *(probabilistic) models*. In this program, we observe that the lawn is wet, which can be caused by either rain or the sprinkler, and try to infer the probability that it was raining. The prior is `bernoulli 0.2` and the likelihood is:

$$w(\texttt{True}) = 0.1 * 0.99 + 0.9 * 0.70 = 0.729$$
$$w(\texttt{False}) = 0.1 * 0.90 + 0.9 * 0.01 = 0.099$$

Given the prior distribution prior_c and the likelihood w_c of a computation `c :: m a`, the distribution μ_c that `c` represents is given by the rescaling of the prior by the likelihood, assigning to each event E the weighted sum:

$$\mu_c(E) := (w_c \odot \text{prior})(E) := \int_E w_c(x)\text{prior}_c(dx)$$

i.e., Lebesgue integrating the likelihood over E according to the prior. Probabilistic programs give us a formalism for expressing distributions by programming with the abstractions of Bayesian statistics: *sampling* from the prior distribution and changing the likelihood, also known as *conditioning*.

The example above is discrete so the integral amounts to a weighted sum:

$$\mu_c(\texttt{True}) = 0.2 * 0.729 = 0.1458$$
$$\mu_c(\texttt{False}) = 0.8 * 0.099 = 0.0792$$

The value of μ_c on the entire domain is called the *model evidence*. Bayesian statisticians regard model evidence that is close to 0 as indicating a bad fit of the model to the data. In general, they regard the model evidence as a score for the whole model. The model evidence of our example is:

$$\mu_c(\texttt{Bool}) = \mu_c(\texttt{True}) + \mu_c(\texttt{False}) = 0.1458 + 0.0792 = 0.225$$

Beyond the model evidence, we are interested in the *posterior* distribution, the probability distribution on program outputs conditioned on the observed data. It is given by normalising μ_c by the model evidence $\mu_c(a)$. Thus in the example above we have:

$$\text{posterior}_c(\texttt{True}\ ) = \texttt{0.1458 / 0.225} = \texttt{0.648}$$
$$\text{posterior}_c(\texttt{False}) = \texttt{0.0729 / 0.225} = \texttt{0.352}$$

Therefore from the observation that the lawn is wet we estimate it rained with probability `0.648`.

Approximate Bayesian inference deals with approximations to the model evidence and the posterior distribution. Traditionally, such inference is done by hand. The descriptions of the prior distribution and the likelihood are constructed manually, the equations of the inference algorithm are derived with pen and paper, and implemented as a module of the intended application. In an attempt to partially automate this process, many libraries automatically derive selected inference algorithms from a suitable intermediate representation of the model. This representation is usually constructed by the user directly as a data structure in some programming language. This approach is used in systems such as Infer.NET [Minka et al. 2014] and PyMC [Patil et al. 2010].

To relieve users from the burden of manually constructing the intermediate representation, *probabilistic programming* systems such as BUGS [Gilks et al. 1994], Stan [Carpenter et al. 2017], and LibBi [Murray 2013] provide a special-purpose modelling language, usually a mixture of C-like syntax and mathematical notation, which is human-readable and from which a suitable compiler automatically generates the required simulations that approximate the model evidence or sample predictions from the posterior distribution. While users express their models as probabilistic programs, as we did above, the expressive power of the languages they use is limited, and the inference process cannot be directly incorporated into larger applications, which resort to external, file-based communication.

Here, we focus on probabilistic programming systems that extend a fragment of an existing programming language with sampling and conditioning. Doing so allows both the users and the implementers to repurpose the existing software development infrastructure for Bayesian inference, but Bayesian inference for such programs is usually more challenging than for programs written in restrictive special-purpose languages. Examples of such systems include IBAL [Pfeffer 2001], Church [Goodman et al. 2008], Anglican [Wood et al. 2014], WebPPL [Goodman and Stuhlmüller 2014], Venture [Mansinghka et al. 2014], and R2 [Nori et al. 2014].

In this paper we describe a Haskell library constituting a probabilistic programming system that extends an existing language with probabilistic effects. Specifically, it provides a monadic typeclass with probabilistic effects that can be used to construct probabilistic programs using arbitrary pure Haskell code. The modelling language is thus expressive, constituting what is sometimes called a *universal* probabilistic programming language [Goodman et al. 2008], like many of the languages described in the paragraph above. The main novelty of our library is its compositional approach to specifying Bayesian inference algorithms. The library is called MonadBayes and it is freely available online[1].

One of the goals of probabilistic programming is to separate probabilistic modelling from inference. The idea is that domain experts can specify probabilistic models according to their knowledge of the underlying processes and then an automated inference engine makes probabilistic inferences using such a model and a record of observations made. While this is feasible to some extent, it is not possible to fully automate Bayesian inference. Indeed, in general the posterior is not computable [Ackerman et al. 2011] and in practice there is a diverse set of approximate inference algorithms targeting different classes of models. It is therefore desirable for probabilistic programming systems

[1]https://github.com/adscib/monad-bayes

$$\begin{tikzcd}
\boxed{\text{Model}} \arrow[r, mapsto] & \boxed{\text{IR}_1} \arrow[r, mapsto, "t_1"] & \boxed{\text{IR}_2} \arrow[r, mapsto, "t_2"] & \cdots \arrow[r, mapsto, "t_{n-1}"] & \boxed{\text{IR}_n} \arrow[r, mapsto] & \boxed{\text{Sampler}}
\end{tikzcd}$$

Fig. 1. Conceptualisation of Bayesian inference. The model is written by the user in a probabilistic programming language and inference produces a sampler, which is a program implementing the selected inference algorithm for the specified model. Inference in most existing systems consists of a single conceptual step, while we can further decompose it into multiple passes through intermediate representations much like traditional compilers do. Crucially, the approximation is contained to the step of executing the final samplers, while all the intermediate transformations are exact.

to provide the users with convenient ways of specifying and extending the available inference algorithms. This task is sometimes called *inference programming* and was recognised by systems such as Venture [Mansinghka et al. 2014], Figaro [Pfeffer 2015], Edward [Tran et al. 2017], and Pyro.

In our recent work [Ścibior et al. 2018] we have developed a denotational semantics capable of expressing and validating the implementation of popular inference algorithms. Our account allows these implementations to be freely combined while ensuring correctness. The semantic implementation uses monad-transformer-lke technique in the recently proposed category of *quasi-Borel spaces* [Heunen et al. 2017]. In this paper we use this semantic development as a basis for an executable modular implementation of a probabilistic programming system where inference algorithms can be easily extended and combined. Furthermore, we implement it as a lightweight library rather than a stand-alone language. We implement the constructions described by Ścibior et al. [2018], providing algorithms such as Lightweight Metropolis-Hastings (LMH) [Wingate et al. 2011], Sequential Monte Carlo (SMC) [Wood et al. 2014], and Resample-Move SMC (RM-SMC) [Gilks and Berzuini 2001]. We also extend it further providing implementations of algorithms such as Particle Marginal Metropolis-Hastings (PMMH) [Andrieu et al. 2010] and SMC^2 [Chopin et al. 2013]. Thanks to the modular construction the resulting Haskell library is compact, with the whole system implemented in about 1200 lines of code. The implementation only relies on common functional programming features, such as higher-order functions, inductive datatypes, and type-classes or expressive module systems so it can be easily ported to other functional languages. Finally, the implementation closely follows its theoretical foundations, giving users high assurance in the correctness of inference algorithms they build using the library. Since the semantic construction is beyond the scope of the present development, we only sketch it informally here and refer the interested reader to Ścibior et al. [2018] for a complete account.

Figure 1 depicts a high-level view of Bayesian inference. We transform the model, containing both sampling and conditioning operations, into a probabilistic program, the *sampler*, containing only sampling operations. While the model is non-executable due to the conditioning operations, the sampler is executable. We can therefore run the sampler and, with some post-processing, approximate the posterior distribution of the original model. The main novelty in our approach is to decompose the inference step, framed in the figure, into a sequence of intermediate steps. These steps, which statisticians only communicate informally, consist of inference-specific *transformations* t_i between inference-specific *representations* IR_i.

Our library provides such inference representations, and inference *transformers* IT, that manipulate these representations. Users can then define custom IRs by composing transformers to obtain inference transformer *stacks*:

$$\text{IR'} = \text{IT}_1 \circ \text{IT}_2 \circ \cdots \circ \text{IT}_\ell \text{IR}$$

Transformers also allow *lifting* an inference transformation to an inference transformation between the transformed representations:

```
lift :: (IR1 → IR2) → IT IR1 → IT IR2
```

Thus inference transformers provide an additional dimension of modularity, allowing us to construct inference representations by combining inference transformers.

Our aim is to enable such lean and flexible high-level probabilistic programming libraries to form viable alternatives to manual Bayesian inference and to existing monolithic probabilistic programming systems. To achieve it, we need a performant implementation allowing us to enjoy the flexibility in model construction and modular inference design without sacrificing performance. We therefore benchmark our library against existing systems and demonstrate competitive performance.

In the evaluation we distinguish between two kinds of efficiency we associate with inference algorithm implementations. The *statistical efficiency* measures the number of iterations needed by the resulting probabilistic sampler to approximate the posterior distribution well. The *computational efficiency* measures the time it takes to execute one interation of the resulting sampler.

Here we are not concerned with improving statistical efficiency, i.e., we do not develop new inference algorithms. We only replicate what already exists in the literature, and leave this important task to expert algorithmic statisticians. Our evaluation focuses on the computational efficiency and we demonstrate that our implementation has similar performance to existing systems of comparable scope. However, profiling shows that the abstractions we use come with significant computational overhead so the computational efficiency of our approach could be further improved, possibly through the application of well-known optimisation techniques. For example, in our preliminary experiments the bottleneck was an inefficient implementation of a free monad. Replacing the free monad with a Church-encoded version removed this performance bottleneck. We expect that applying similar techniques, such as finally-tagless [Carette et al. 2009], stream fusion [Coutts et al. 2007], information-flow graphs and other static-analyses, alongside aggressive compiler optimisations, would remove similar bottlenecks.

This expectation is unsurprising once we notice that Figure 1 depicts the typical structure of an optimising compiler, transforming the source code, the model, through a series of intermediate representations, and emitting an optimised object code, the sampler. Our denotational account [Ścibior et al. 2018] also fits with this perspective, as a common way to validate compiler transformations is by validating that the translations preserve the semantics of the transformed program. Therefore, we view our approach and its associated libraries as a vehicle to open up a fruitful application area for programming language experts in general, and functional programmers in particular. In particular we hope that by breaking down implementations of approximate Bayesian inference algorithms into smaller components with well-defined semantics, our work will make it easier for experts in implementations of programming languages to contribute to the field of probabilistic programming even if they do not posses an in-depth knowledge of probability and statistics. We envision such contributions would mainly come in the form of reducing the computational overhead of the abstractions we present in this paper, such as the ones mentioned in the paragraph above.

We prefix our development with two remarks for Bayesian inference cognoscenti. First, we focus on the most general family of inference algorithms, namely samplers that do not use *gradient* information. We ignore: gradient-based techniques such as Hamiltonian Monte Carlo, that assume the likelihood function is differentiable; optimisation based methods, including variational inference; and enumeration based methods such as belief propagation. While these classes of algorithms are less generic than sampling, they are still important and we plan to develop them in future work.

Second, our library implements basic versions of advanced sampling algorithms. However, their successful application in practice requires incorporating established heuristics, such as: adaptive proposal distributions, controlling resampling with effective sample size, tuning rejuvenation kernels based on population in SMC^2, and so on. We believe these are largely orthogonal to the core design, so excluding them makes for a clearer and more accessible presentation of the main ideas.

The paper is structured as follows. In Section 2 we show the construction of basic inference transformers and associated transformations. Section 3 demonstrates how to use those basic building blocks to construct more advanced building blocks that express simple inference algorithms. In Section 4 we show how to compose those building blocks to obtain advanced inference algorithms. Then in Section 5 we present an empirical evaluation of our library comparing it with existing probabilistic programming systems in terms of performance and implementation effort required. In Section 6 we discuss how our modular structure enables a deterministic approach to testing inference algorithm implementations. In Section 7 we outline an ongoing OCaml implementation of our design, and demonstrate how an expressive module system can replace our use of type-classes. Finally, in Section 8 we discuss related work and in Section 9 we present directions for future work and conclude.

Our target audience are functional programmers and language implementers that are interested in making their languages viable platforms for statistical machine learning and data science. All sections, with the exception of Section 4, have this target audience in mind. We do not expect this audience to easily see the reasons why the programs we describe in these sections work, such developments can fill a standalone article [Ścibior et al. 2018]. Instead, we want to demonstrate to this audience that our library code uses the usual programming abstractions functional programmers are used to code and optimise.

However, our goal is to allow probabilistic programming experts to express their inference algorithms as close to their specialist language as possible. We demonstrate this ability in Section 4, and this section alone is intended for Bayesian inference specialists. Without this section, our account is unlikely to convince a probabilistic programming expert that we can indeed express state-of-the-art algorithms. Non-experts may want to only briefly skim this section to get an impression of typical client code for our library.

2 BASIC BUILDING BLOCKS

We express the building blocks for inference algorithms in Haskell. Unless stated otherwise, the structures presented in this section follow the mathematical formulation of Ścibior et al. [2018]. We distinguish three types of building blocks:

(1) **Inference representations**
Inference representations are data structures representing distributions. Concretely, they are instances/implementations of the monad type-class/interface, but they need *not* satisfy the monad laws. We use them as the intermediate representation in the inference/compilation process.
In Haskell, we express these abstract interfaces as type-classes, as in Figure 2. There are three separate type-classes: the *sampling* representation `MonadSample`, and the *conditioning* representation `MonadCond`. A representation that is both a sampling and conditioning representation is called an *inference* representation `MonadInfer`. Together with the `Monad` interface, the interfaces in the figure can express our probabilistic programs of interest. Our library includes default implementations for common probability distributions in terms of `random`, which is a uniform distribution on the unit interval $[0, 1]$. Specific representations can overwrite these default implementations for better efficiency.

```
type R = Double
class Monad m ⇒ MonadSample m where
  random :: m R
  bernoulli :: R → m Bool
  bernoulli p = fmap (< p) random
  -- and other default distributions:
  -- normal, gamma, beta, geometric,
  -- poisson, dirichlet
class Monad m ⇒ MonadCond m where
  score :: Log R → m ()
class (MonadSample m, MonadCond m) ⇒ MonadInfer m
```

Fig. 2. Inference representations using Haskell type-classes. `Log R` is a numeric type representing non-negative real numbers using their logarithms.

To reason about correctness of algorithms involving a representation `m`, one defines a semantics map μ `:: m a` $\to$ `D a` assigning to each representation `c :: m a` a distribution over its return type μ_c `:: D a`. This "type" of distributions and the semantics map are mathematical objects impossible to implement. They are pure reasoning abstractions, and to ensure correctness, we require the semantics map preserves the monadic structure. Since `D` is itself a monad, both in terms of an interface and satisfying the monad laws, we require that the following equations hold.

$$\mu\ .\ \texttt{return} = \texttt{return} \qquad\qquad \mu\ (\texttt{c >>= f}) = \mu(\texttt{c})\ \texttt{>>=}\ (\mu.\ \texttt{f})$$

Furthermore, we require that μ `(random)` is the uniform distribution over the unit interval $[0, 1]$, and that μ `(score r)` is the (unique) distribution over the singleton with total measure `r`. See Ścibior et al. [2018] for the full details.

(2) **Inference transformations**
Inference transformations are mappings `t :: m a` $\to$ `m' a` between inference representations `m`, `m'`. They can be thought of as passes in a compilation process. For reasoning purposes, such an inference transformation is *correct* if it preserves the semantics map of the transformed inference representation: μ `. t` $= \mu$. It follows that a composition of correct inference transformations is a correct inference transformation, so algorithms given by composition of correct inference transformations are correct by construction. Unless stated otherwise, all inference transformations are correct based on the reasoning in Ścibior et al. [2018]. As a consequence, the weighted samplers our inference algorithms produce are *unbiased* by construction. Being unbiased means that the weighted expectation of the sampler is identical to the expectation of the model.

(3) **Inference transformers**
Inference transformers are compositional building blocks of inference representations, much like monad transformers are building blocks of monads. By analogy with monad stacks we call a sequence of inference transformers an *inference stack* and implicitly identify a stack $[\mathtt{mT}_1, \mathtt{mT}_2, \ldots, \mathtt{mT}_n]$ with the composed transformer $\mathtt{mT}_1\ .\ \mathtt{mT}_2\ .\ \ldots\ .\ \mathtt{mT}_n$. Our inference transformations often apply to a specific transformer in the stack, and are polymorphic in the remainder of the stack. Like monad transformers, inference transformers specify a function to lift a transformation through them, which we call `hoist` since `lift` is already used in Haskell. These abstractions allow us to apply inference transformations to particular layers in the inference stack.

This section uses these abstractions to define basic inference building blocks. These basic blocks do not express advanced algorithms by themselves, but abstract away details to expose the structure of more advanced building blocks. The subsequent Section 3 presents two such building blocks, and the following Section 4 shows how to compose them to obtain advanced inference algorithms.

2.1 Models

The following example program implements a simple random walk using the abstractions introduced above. It models a particle travelling in one dimension in discrete time. At each time step the particle moves randomly according to a Gaussian distribution. The model takes as argument a data-set containing noisy observations about the particle's location. The model captures our hypothesis about the particle's movement, subject to some unknown parameters such as the rate in which it is moving. The goal of inference is to fit this model to the observed data, updating these parameters accordingly.

```
1   random_walk :: MonadInfer m ⇒ [R] → m [R]
2   random_walk ys = do
3     s ← gamma 1 1
4     let expand xs [] = return xs
5         expand (x:xs) (y:ys) = do
6           x' ← normal x s
7           obs x' y
8           expand (x':x:xs) ys
9         where
10          obs x y = score (normalPdf x 1 y)
11    xs ← expand [0] ys
12    return (reverse xs)
```

The *parameter* s controls the deviation of the particle from its current position. We emphasise that the parameter in this case is the sampling operation in the dynamic call to gamma on line 3, rather than the static program variable s. The model starts at position 0, and iteratively constructs the desired list of locations in reverse. At each step, we sample the next location x' from a normal distribution around x with a standard deviation of s (line 6). This sample is then fitted to the observed data point y, using the helper function obs from line 10. It uses the probability density function (pdf) for the normal distribution

$$\texttt{normalPdf}\ \mu\ \sigma\ z := \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(z-\mu)^2}{2\sigma^2}}$$

centered around x with standard deviation 1, representing our assumption that y is a noisy observation of x, where the noise distributes normally with standard deviation 1. The call obs x' y on line 7 thus lowers the score of samples x' that are far from the observed location y. In standard statistical modelling terminology the calls to normal x s on line 6 are called *latent variables*. The output of the model is the predicted sequence of true positions of the particle based on provided noisy observations. It is reversed only because Haskell lists are more easily extended at the front than at the back so expand constructs this list in the reverse order.

The type of random_walk is abstract with respect to the inference representation m. Thus we express models as computations constructed in terms of an abstract inference representation interface. This architecture lends itself to a shallowly embedded probabilistic programming DSL, with the usual benefits of DSLs, e.g., the ability to call standard library functions such as reverse. We could alternatively use a stand-alone language for expressing models that a suitable front-end would convert to a desired inference representation.

2.2 Basic Samplers

To interpret the program above we need to construct a concrete inference representation. The simplest one is a sampler that draws concrete values for random variables from the prior. Such a sampler type can be constructed as a state monad that references a global pseudo-random number generator. Since the generator is mutable, in Haskell we need to use the `ST` monad.

```
1  newtype Sampler a =
2    Sampler (forall s. ReaderT (GenST s) (ST s) a)
3  instance MonadSample Sampler where
4    random = Sampler $ do
5      gen ← ask
6      lift (MWC.uniform gen)
```

Line 5 retrieves the reference to the random seed, line 6 uses the standard library `MWC` for random number generation, and lifts it to the underlying reader monad transformer.

We can directly execute computations of type `Sampler` to obtain concrete samples, for example:

```
runSampler :: Sampler a → a
runSampler (Sampler s) = runST $ do
  gen ← MWC.create
  runReaderT s gen
```

The represenation `Sampler` is a sampling representation, i.e., an instance of `MonadSample`, but not a conditioning representation, i.e., an instance of `MonadCond`: `Sampler` does not support conditioning. To obtain a working inference algorithm we need to add an interpretation of `score`, by applying a suitable inference transformer.

We use the *weighting* inference transformer `W`. Theoretically it is the writer monad transformer for the multiplicative monoid structure on `Log R` given by multiplication, but in our implementation we are using the state monad transformer instead since we found it to be much faster. In either case, `W m a` is an `m`-computation returning pairs `(a, Log R)` of the result type together with the accumulated log-likelihood.

```
newtype W m a = W (StateT (Log R) m a)
runW :: Monad m ⇒ W m a → m (a, Log R)
runW (W x) = runStateT x 1
instance MonadSample m ⇒ MonadSample (W m) where
  random = lift . random
instance Monad m ⇒ MonadCond (W m) where
  score w = W (modify (* w))
hoistW :: (forall x. m x → n x) → W m a → W n a
hoistW f (W m) = W (mapStateT f m)
```

Weighting a representation `m` equips it with conditioning operation making `W m` a conditioning representation. If `m` was a sampling representation, its weighted version is also a sampling representation by lifting the sampling operation. Finally, the function `hoist` lifts inference transformations applicable to `m` and turns them into an inference transformations applicable to `T m`. When `m` is already a conditioning representation, we may use the conditioning available through the weighting transformer, or hoist `score r` to use the ambient conditioning of `m`.

We construct a simple inference algorithm by interpreting the model in `W Sampler a` and unwrapping type constructors to obtain a *weighted sampler* of type `Sampler (a, Log R)`. When `a` is a numeric type, we can approximate the expectation of the model by repeatedly running the sampler and calculating the weighted average. Unfortunately such an algorithm usually has low statistical efficiency and it needs to generate impractically many samples to obtain good predictions. Therefore, advanced inference algorithms first apply multiple inference transformations between more advanced inference representations before arriving at the final weighted sampler representation.

2.3 Population

The Pop inference transformer turns a single sample into a collection of weighted samples called the *population*. It is the weighted list transformer, i.e., the composition of W with the ListT transformer.

```
newtype Pop m a = Pop (W (ListT m) a)
  deriving(Monad,MonadSample,MonadCond,MonadInfer)
runPopulation :: Monad m ⇒ Pop m a → m [a, Log R]
runPopulation (Pop p) = runListT (runW p)
hoistP :: (forall x. m x → n x) → Pop m a → Pop n a
hoistP f (Pop m) = Pop (hoistW (mapListT f) m)
```

The collection of samples in the population is usually referred to as *particles*. The usual problem with treating ListT as a monad transformer, namely that ListT-transformed monads do not always satisfy the monad laws, does not apply to inference representation: we do not require that our representations satisfy the monad laws.

In this paper we use three inference transformations associated with Pop:

```
spawn    :: Monad       m ⇒ Int     → Pop m ()
resample :: MonadSample m ⇒ Pop m a → Pop m a
pushEvidence :: MonadInfer m ⇒ Pop m a → Pop m a
```

One is (spawn n >>), which increases the population size n times adjusting the weights accordingly. Next comes resample, which draws a new population with uniform weights from the current population. Resampling's purpose is to remedy situations when a single sample has a large weight compared to the other particles in the population and dominates the result making the other particles irrelevant. Finally, we have pushEvidence which normalizes the weights in the population, while at the same time incorporating the sum of the weights as a score in m.

The meaning function μ is defined in terms of the weighted average over the population. We can state it concisely by making use of pushEvidence and a categorical distribution.

$$\mu^{\mathrm{Pop\ m}}(\mathrm{p}) = \mu^{\mathrm{m}}(\texttt{runPopulation (pushEvidence p) >>= categorical})$$

In the above categorical is a distribution that draws a sample from a weighted list with probabilities proportional to weights.

Since all of the presented transformations preserve the weighted average μ, they are correct inference transformations. The details of resample may vary since there are multiple good choices. Our library uses systematic resampling [Doucet and Johansen 2011] due to its good computational efficiency.

By itself Pop is similar to W. To appreciate its utility we need to combine it with different inference transformers, where it abstracts away the maintenance of the particle population.

3 ADVANCED BUILDING BLOCKS

Sequential Monte Carlo (SMC) and Markov Chain Monte Carlo (MCMC) are two of the most general inference algorithms for probabilistic programs. When we express them as inference representations and reuse the basic building blocks of the previous section, we obtain the components that underlie many advanced inference algorithms.

3.1 Sequential

Many models exhibit a sequential structure where observations are interleaved with sampling. In those models a possible inference strategy is to consider a program up to a certain point, do inference on the partial posterior it defines, then run the program a little more, do more inference, and so on. To implement such algorithms we introduce the sequential transformer `Seq` which introduces suspensions after each `score` in the program. `Seq` is the standard coroutine transformer [Blažević 2011]. In our library we use the implementation of the coroutine transformer available in the monad-coroutine library[2] but the snippet below shows how to implement it from scratch.

```
data Seq m a = Seq (m (Either a (Seq m a)))
runSeq :: Seq m a →
m (Either a (Seq m a))
runSeq (Seq m) = m
instance Monad m ⇒ Monad (Seq m) where
  return x = Seq (return (Left x))
  Seq c >>= f = Seq $ do
    t ← c
    case t of
      Left x → runSeq (f x)
      Right m → return (Right (m >>= f))
```

```
instance MonadTrans Seq where
  lift = Seq . fmap Left
suspend :: Monad m ⇒ Seq m ()
suspend = Seq (return
                (Right (return ())))
instance MonadSample m ⇒
MonadSample (Seq m) where
  random  = lift random
instance MonadCond   m ⇒ MonadCond
(Seq m) where
  score w = lift (score w) >> suspend
```

We have two inference transformations associated with `Seq`:

```
advance :: Monad m ⇒ Seq m a → Seq m a
advance (Seq m) = Seq (m >>= either (return . Left) runSeq)
finish :: Monad m ⇒ Seq m a → m a
finish  (Seq m) = Seq (m >>= either return finish)
```

The `advance` transformation runs the program to the next suspension point. The `finish` transformation runs the program to the end. When reasoning about `Seq`, the meaning function is:

$$\mu^{\texttt{Seq m}}(c) := \mu^{\texttt{m}}(\texttt{finish c})$$

Finally, `hoistS` applies the inference transformation only to the part of the program executed so far.

```
hoistS :: (forall x. m x → m x) → Seq m a → Seq m a
hoistS tau (Seq m) = Seq (tau m)
```

Combining `Seq` with `Pop`, we obtain a Sequential Monte Carlo variant known as the *particle filter* [Doucet and Johansen 2011] that we refer to simply as SMC. Within the context of SMC, recall that a sample in a population is called a *particle*. The algorithm starts by initialising a population of size `n`, then repeatedly runs the program to the next `score`, resamples the population, runs to the next `score` and so on. We implement it by composing the inference transformations we have introduced so far.

```
smc :: MonadSample m ⇒ Int → Int → Seq (Pop m) a → Pop m a
smc k n =  finish . compose k (advance . hoistS resample) . hoistS (spawn n >>)
```

The argument `k` is the number of time steps in SMC, `n` is the number of particles used, and

```
compose :: Int → (a → a) → a → a
```

is `k`-fold function composition. To execute the sampler, we use the instance where `m` is `Sampler`.

[2]http://hackage.haskell.org/package/monad-coroutine

3.2 Traced

The final transformer we present supports a class of algorithms known as *Trace Markov Chain Monte Carlo (MCMC)*, which mix two ingredients. The first is the general Metropolis-Hastings update which underlies the MCMC approximation technique. The second ingredient is a representation of the *traces* of the program, the trace being a record of random choices the program has made.

The idea behind MCMC is to represent a distribution over a space `a` as a simulation of a random walk through the space `a` according to some predefined *proposal kernel* `k :: a → m a`. If the simulation is currently at value `x`, the kernel `k` determines a distribution `k x` over the proposed values. The second fundamental part of MCMC is the *rejection rate* ρ. This rate is a non-negative function ρ `: a → a → Log R`. At each step, given some `x :: a` sampled from a distribution `c :: m a`, we sample a new proposal `y∼k x` and, with probability ρ `x y`, we *accept* the new proposal `y`, taking it as the new point, or *reject* it and remain with `x`. We can summarise this process in the following code:

```
abstractMH :: MonadSample m ⇒ m a → m a
abstractMH c = do
  x ← c
  y ← k x
  b ← bernoulli ρ x y
  if b then return y
       else return x
```

If the kernel `k` and the rejection rate ρ are correctly chosen, then `abstractMH` is a bona fide inference transformation. The mathematical justification for this fact is the Metropolis-Hastings-Green theorem.

Choosing a proposal kernel and rejection rate that would work for all types `a` is a tall order. Instead, the Trace MCMC family of algorithms replaces these types with traces through the probabilistic model we want to infer. For a concrete example, consider the sprinkler model from the introduction. The trace would contain the values sampled for the variables `rain` and `sprinkler`. The Trace MCMC algorithm maintains a distribution over such traces, using a suitable proposal kernel to generate a new trace through the model.

Here, we take traces to be lists of real numbers `[R]` from the unit interval [0, 1], each corresponding to one invocation of `random` in the program. For example, in the sprinkler model a trace `[0.15, 0.5]` would correspond to `rain = True` and `sprinkler = False`, while `[0.3, 0.05]` would correspond to `rain = False` and `sprinkler = True`.

If the variable names in the program are globally unique, we could have a record where each field corresponds to the random variable with the same name. However, in a general model, samples may be nested in complicated control flow. Wingate et al. [2011] devised a popular method for tagging random choices in the program based on the context in which they are executed. The random choices stored in the trace are often augmented with additional information such as the distribution they were drawn from or some control flow information extracted from the program [Mansinghka et al. 2014]. These sophisticated representations of traces can improve the statistical efficiency of MCMC algorithms. They are compatible with the design we present in this paper but we refrain from using them for simplicity. We believe that the static analysis community has good general representations of program traces to offer to the probabilistic programming community.

A traced inference representation consists of two components, one being the trace and the other a representation that can run the full program with a modified trace. There are multiple possible ways to combine these two components, which trade off computational efficiency for flexibility. Below we present a sequence of `Tr` datatypes, each more expressive than the previous in a sense of allowing additional inference transformations, but less computationally efficient in cases where the additional flexibility is not required. The first one was originally presented by Ścibior et al. [2018], while the subsequent ones are novel extensions of it. We expect the additional inference transformations associated with these extensions to be correct, although we have not proven that in the framework of Ścibior et al. [2018].

3.2.1 Full Tracing of the Whole Program. We begin with the most straightforward construction that allows our basic modular implementation [Ścibior et al. 2018] of the Trace Metropolis-Hastings algorithm. It consists of a weighted free monad over `random` and a computation generating a trace in the transformed inference representation. For efficiency we use the Church-encoded version of the free monad from the package free[3], that is `F f = forall r. (a → r) → (f r → r) → r`.

```haskell
-- sampling functor
newtype SamF a = Random (R → a)

data Tr m a = Tr (W (F SamF) a)
                 (m ([R], a))
traceDist (Tr m d) = d
marginal :: Monad m ⇒ Tr m a → m a
marginal (Tr m d) = fmap snd d

instance Monad m ⇒ Monad (Tr m) where
  return x = Tr (return x)
                (return ([],x))
  (Tr mx dx) >>= f = Tr my dy where
    my = mx >>= model . f
    dy = do
      (us, x) ← dx
      (vs, y) ← traceDist (f x)
      return (us <> vs, y)
```

```haskell
instance MonadSample m ⇒
MonadSample (Tr m) where
  random =
    Tr random
       (fmap (\u → ([u],u)) random)

instance MonadCond m ⇒
MonadCond (Tr m) where
  score w = Tr (score w)
               (score w >> pure ([],()))

hoistT :: (forall x. m x → m x)
          → Tr m a → Tr m a
hoistT f (Tr m d) = Tr m (f d)

mhStep :: MonadSample m ⇒ Tr m a → Tr m a
mh :: MonadSample m ⇒
        Int → Tr m a → m [a]
```

For efficiency we store the output of the program along with the trace. The implementation of `mhStep` follows the structure of `abstractMH` above, suitably instantiated for traces. We emphasise that `Tr` is not an instance of `MonadTrans` since it does not allow for computation in `m` to be lifted to `Tr m`. For reasoning, the semantic function of `Tr` is defined in terms of `marginal`, which marginalises the trace and the model, leaving only the return value:

$$\mu^{\texttt{Tr m}}(\texttt{c}) := \mu^{\texttt{m}}(\texttt{marginal c})$$

The inference transformation `mhStep` performs a single step of the Trace MH algorithm updating the trace but leaving the program unchanged. Specifically, a new trace is proposed by taking the old trace and randomly modifying one of the random variables in it, selected again at random. Since the number of random variables used in the program can vary dynamically, the length of the new trace is adjusted to match the length required by the program. If the trace is too long it is truncated, if it is too short it is extended with freshly sampled values. This adjustment requires a pass through the program so at the same time we compute the likelihood associated with the adjusted trace. Finally, based on the ratio of likelihoods, as well as some correcting factors [Wingate et al. 2011], we compute the probability of accepting the new trace. With that probability we retain the proposed trace, otherwise we keep the old one.

Repeating this procedure multiple times defines a Markov process on the space of execution traces, which constitutes the Trace MH algorithm. It is available in our library as the `mh` inference

[3]http://hackage.haskell.org/package/free

transformation. However, the basic building block is `mhStep`, forming a component of larger inference algorithms such as the resample-move SMC in Section 4.1.

The efficiency of MH crucially depends on the choice of the proposal kernel. Our library uses as a default the single-site kernel sampling from the prior as proposed by Wingate et al. [2011].

3.2.2 Partial Tracing of the Whole Program. The construction presented above is suitable if all random variables in the program are subject to the Trace MH updates. This is not the case in the family of inference algorithms known as pseudo-marginal MH, where only a subset of variables is updated using MH and the remaining ones are marginalized using another inference algorithm. This marginalization is usually performed approximately using importance sampling. This is the case for all the algorithms we present in this paper, but the marginalization could also be done with enumeration or a different inference algorithm.

To enable pseudo-marginal MH methods we extend the construction above by replacing the free monad with a free monad transformer applied to `m`. This transformer enables us to lift computations in `m` into `Tr m`, which is not possible with the previous construction. In Haskell this change means `Tr` becomes an instance of the `MonadTrans` class. The variables in computations lifted from `m` are then marginalized by `m` as far as Trace MH is concerned while the ones created in `Tr m` are subject to Trace MH updates. In Section 4.2 we present a concrete pseudo-marginal inference algorithm constructed in this fashion.

```
data Tr m a = Tr (W (FT SamF m) a)
                 (m ([R], a))
```

```
instance MonadTrans Tr where
  lift m = Tr (lift $ lift m)
              (fmap (\x → ([],x)) m)
```

Note that we define `lift` in such a way that the lifted random variables are resampled at every `mhStep`. If we only included `m` in the second component of `Tr` and not the first, they would be fixed throughout MH updates. While that is also potentially useful, this is not what pseudo-marginal MH requires so we do not pursue this possibility here. We do not show code for the remaining instances and transformations since it is exactly the same as for the version above.

3.2.3 Partial Tracing of Program Fragments. In certain situations it is desirable to freeze the values of random variables using the contents of the current trace. This is useful when we know we will not update them any more but still want to keep the `Tr` structure for the random variables that come later. We present a concrete use case for this operation in Section 4.1.

```
-- sampling functor
newtype SamF a = Random (R → a)
-- withTrace runs the program using
-- the trace
withTrace :: [R] → FT SamF m a → m a
discardWeight :: W m a → m a

data Tr m a = Tr (m (W (FT SamF m) a,
                        ([R], a)))
runTr (Tr c) = c
marginal :: Monad m ⇒ Tr m a → m a
marginal = fmap (snd . snd) . runTr
```

```
instance Monad m ⇒ Monad (Tr m) where
  return x = Tr (return (return x,
                          ([],x)))
  (Tr cx) >>= f = Tr $ do
    (mx, (tx, x)) ← cx
    let m = mx >>=
      join . lift . lift .
      fmap fst . runTr . f
    (_, (ty, y)) ← runTr $ f x
    let t = tx ++ ty
    return (m, (t, y))
```

```
instance MonadTrans Tr where
  lift m = Tr $ fmap
    (\x → (lift $ lift m, ([],x))) m

instance MonadSample m ⇒
MonadSample (Tr m) where
  random = Tr $ fmap
    (\u → (random, ([u], u))) random

hoistT :: (forall x. m x → m x)
            → Tr m a → Tr m a
hoistT f (Tr c) = Tr (f c)
```

```
instance MonadCond m ⇒
MonadCond (Tr m) where
  score w = Tr $ fmap
    (\x → (score w, x))
    (score w >> return ([],()))

freeze :: Monad m ⇒ Tr m a → Tr m a
freeze (Tr c) = Tr $ do
  (_, (_, x)) ← c
  return (return x, ([], x))
```

The only difference from the previous construction is that we pushed the weighted free monad into `m`, which lets us implement the `freeze` inference transformation that commits to values stored in the current trace. This is useful if we later extend the program and do not want to update values for some variables anymore. We use it in Section 4.1 to obtain an efficient variant of resample-move SMC.

This implementation is strictly more expressive than the previous two so in principle it could be used instead of them. It is also significantly less computationally efficient due to the additional abstraction layers. We therefore prefer the previous constructions of `Tr` whenever possible.

4 COMPOUND INFERENCE ALGORITHMS

We demonstrate how to build sophisticated inference algorithms by combining MH and SMC in different ways. We implement these algorithms by composing specific inference transformations, each of which is defined for a particular transformer. The implementation of resample-move SMC below follows the construction of Ścibior et al. [2018], while the other implementations are novel. We adhere fully to the inference representation interface, and only compose the basic inference transformations and their hoistings, maintaining the inference representation abstraction. Our library is the first probabilistic programming system that supports this level of compositionality.

4.1 Resample-Move SMC

A common problem with particle filters is that of particle degeneracy, where after resampling many particles are the same, effectively reducing the sample size. One way to ameliorate this problem is to introduce rejuvenation moves, where after each resampling we apply a number of MCMC transitions to each particle independently, thus spreading them around the space. If we use an MCMC kernel that preserves the target distribution at a given step, the resulting algorithm is correct. This algorithm is known as the resample-move SMC (RM-SMC) and was originally introduced by Gilks and Berzuini [2001].

To implement RM-SMC we use the stack `Seq Tr Pop`. Inlining the types, a program is interpreted as a population of traced coroutines. It allows us to apply MH transitions to partially executed coroutines, which is exactly what we require for the rejuvenation steps. The implementation of resample-move SMC is similar to that of SMC, with the introduction of `mhStep`.

```
rmsmc :: MonadSample m ⇒ Int → Int → Int →
           Seq (Tr (Pop m)) a → Pop m a
rmsmc k n t = marginal . finish .
    compose k (advance . hoistS (
        compose t mhStep . hoistT resample)) .
    (hoistS . hoistT) (spawn n >>)
```

In the above t is the number of MH transitions to be applied after each resampling step.

The version of RM-SMC presented above is computationally intensive. In some models it is better to restrict the rejuvenation transitions to the subset of random variables introduced since the last resampling. We can accomplish that using the freeze transformation from Section 3.2.

```
rmsmcLocal :: MonadSample m ⇒ Int → Int → Int →
              Seq (Tr (Pop m)) a → Pop m a
rmsmcLocal k n t = marginal . finish .
    compose k (advance . hoistS ( freeze .
        compose t mhStep . hoistT resample)) .
    (hoistS . hoistT) (spawn n >>)
```

4.2 Particle Marginal MH

RM-SMC uses the MH update inside SMC. An alternative composition is to use SMC inside an MH update. A particular instance is the algorithm called Particle Marginal Metropolis-Hastings (PMMH) [Andrieu et al. 2010], a pseudo-marginal MH algorithm that uses SMC to approximately integrate over the latent variables in the model. It is primarily used for parameter estimation in time series models.

PMMH is only applicable to models with a specific structure, namely the probabilistic program needs to decompose to a prior over the global parameters m param and the rest of the model param → m a. Combining these using >>= would yield the complete model of type m a. For example, the random walk model from Section 2.1 would be decomposed as follows:

```
s :: MonadSample m ⇒ m R
s = gamma 1 1
random_walk' :: MonadInfer m ⇒ [R] → R → m [R]
random_walk' ys s = do
  let obs x y = score (normalPdf x 1 y)
  let expand xs [] = return xs
      expand (x:xs) (y:ys) = do
        x' ← normal x s
        obs x' y
        expand (x':x:xs) ys
  xs ← expand [0] ys
  return (reverse xs)
```

The idea is to do MH on the parameters of the model. Recall that for MH we need to compute the likelihood for the particular values of parameters but that involves integrating over the remaining random variables in the model which is intractable. Fortunately to obtain valid MH it is sufficient to have an unbiased estimator for the likelihood which is produced by a single sample from W. MH with such an estimator is referred to as pseudo-marginal MH. If instead of taking a single weight from W we take the sum of weights from Pop we obtain an unbiased estimator with lower variance. In particular if such a Pop is a result of smc the resulting algorithm is known as PMMH.

The full implementation of PMMH is then as follows:

```
pmmh :: MonadInfer m
    ⇒ Int    -------------  -- t: number of MH steps
    → Int    -------------  -- k: number of time steps
    → Int    -------------  -- n: number of particles
    → Tr m b                -- param: model parameters prior
    → (b → Seq (Pop m) a)   -- model
    → m [[(a, Log R)]]      -- result
pmmh t k n param model =
  mh t (param >>= runPopulation . pushEvidence .
    hoistP lift . smc k n . model)
```

The code above can be read as follows. First it applies SMC to the model and lifts the entire SMC computation through `Tr`. Then it `scores` the sum of weights in `Tr` and keeps the population as the output. This preprocessed computation is combined with the prior on parameters. Running `mh` then produces exactly the desired algorithm, since the only `score` in `Tr` is the sum of weights from the population.

4.3 SMC^2

The final inference algorithm we discuss, proposed by Chopin et al. [2013], can be regarded as a hybrid of resample-move SMC and PMMH approaches and is used for joint estimation of the posterior over parameters and latent variables in state-space models. It features an outer population of particles, much like RM-SMC, each of which holds different values for the parameters. These particles are filtered through observations using resampling and MH-based rejuvenation where appropriate. However, like in PMMH, these MH transitions do not use exact densities but rather estimators obtained from an inner particle filter over the latent variables. The two particle filters are synchronised in the sense that they step through the same observations simultaneously.

Like PMMH, SMC^2 is only applicable to programs separable into the prior over parameters and the rest of the model. Furthermore, for SMC^2 we need a variant of `smc` that performs `pushEvidence` after each step. We call it `smcPush` and its implementation is almost identical to `smc`.

```
smcPush :: MonadInfer m ⇒
  Int → Int → Seq (Pop m) a → Pop m a
smcPush k n =
  finish . compose k (advance .
           hoistS (pushEvidence . resample)) .
  hoistS (spawn n >>)
```

We want to instantiate `m` to `Seq (Tr (Pop Sampler))` and run `rmsmc` on it. Unfortunately doing that naively has the unintended consequence that the random variables from `model` end up being traced which is not what we want. To remedy this situation we introduce a type synonym that performs the necessary lifting.

```
newtype SMC2 m a = SMC2 (Seq (Tr (Pop m)) a)
  deriving(Monad)
setup (SMC2 m) = m
instance MonadTrans SMC2 where
  lift = SMC2 . lift . lift . lift
instance MonadSample m ⇒ MonadSample (SMC2 m) where
  random = lift random
instance MonadCond m ⇒ MonadCond (SMC2 m) where
  score = SMC2 . score
```

The `SMC2` synonym thus ensures that the random variables bypass the transformers that need not be concerned with them. We can then complete the SMC2 implementation as follows.

```
smc₂ :: MonadSample m
     ⇒ Int ------------------------ -- k: number of time steps
     → Int ------------------------ -- n: number of inner particles
     → Int ------------------------ -- p: number of outer particles
     → Int                          -- t: number of MH transitions
     → Seq (Tr (Pop m)) b           -- param: model parameters
     → ( b → Seq (Pop (SMC2 m)) a)  -- model
     → Pop m [(a, Log R)]
smc₂ k n p t param model =
  rmsmc k p t (param >>= setup . runPopulation .
               smcPush k n . model)
```

Our framework allows more complicated compositions of similar kind, for example using RM-SMC within SMC^2 or introducing two types of parameters with varying scope. Their implementation would be analogous to the examples presented above.

5 EVALUATION

To evaluate our architecture, we compare with state-of-the-art probabilistic programming systems Anglican [Wood et al. 2014] and WebPPL [Goodman and Stuhlmüller 2014] since they implement similar inference algorithms and their front-end languages are extensions of popular programming languages, Clojure and Javascript respectively.

Benchmarks. To check if there is a significant overhead associated with the abstractions we compare execution times on a set of popular benchmarks [Tolpin et al. 2015]. The models we use are logistic regression (LR), hidden Markov model (HMM), and a latent Dirichlet allocation (LDA). Each of these models has a parameter that controls the size of the dataset, namely the number of labelled examples for LR, sequence length for the HMM, and document length for LDA. We run the inference algorithms SMC, MH, and RM-SMC from Sections 2 and 4 on these models comparing execution times. For RM-SMC we use the `rmsmcLocal` and compare it with the corresponding WebPPL implementation. We do not compare Anglican here since it currently does not implement RM-SMC.

5.1 Quantitative Evaluation

Figure 3 shows how execution time scales with the size of the dataset. It shows that the cost of both MH and SMC increases linearly with model size in each implementation as expected. The plots show that Anglican and WebPPL have a noticeable starting overhead compared to our library. We expect this overhead to stem from just-in-time compilation in the Java and NodeJS virtual machines.

The slopes of each line are a measure of the time needed to incorporate an additional data point, with steeper slopes corresponding to higher cost. The slopes for different systems are model-dependent and we attribute these differences to a variety of factors, such as differences in: the data structures used in the model; the pseudo-random number generators; and performance characteristics of the host language.

For RM-SMC our implementation scales linearly with dataset size while WebPPL appears to scale quadratically. We suspect this is due to WebPPL implementation traversing the whole trace at MH updates, even though only a fixed number of variables at the end are candidates for updates. We avoid this issue using the `freeze` transformation and achieve linear scaling.

Figure 4 shows how execution times scale with the number of samples produced or the number of MH transitions performed. In all cases the scaling is linear as expected. For MH and SMC the slopes of all lines for the three systems are similar which indicates that the cost of an additional sample is similar, although Anglican and WebPPL again suffer from some initial overhead. For RM-SMC the

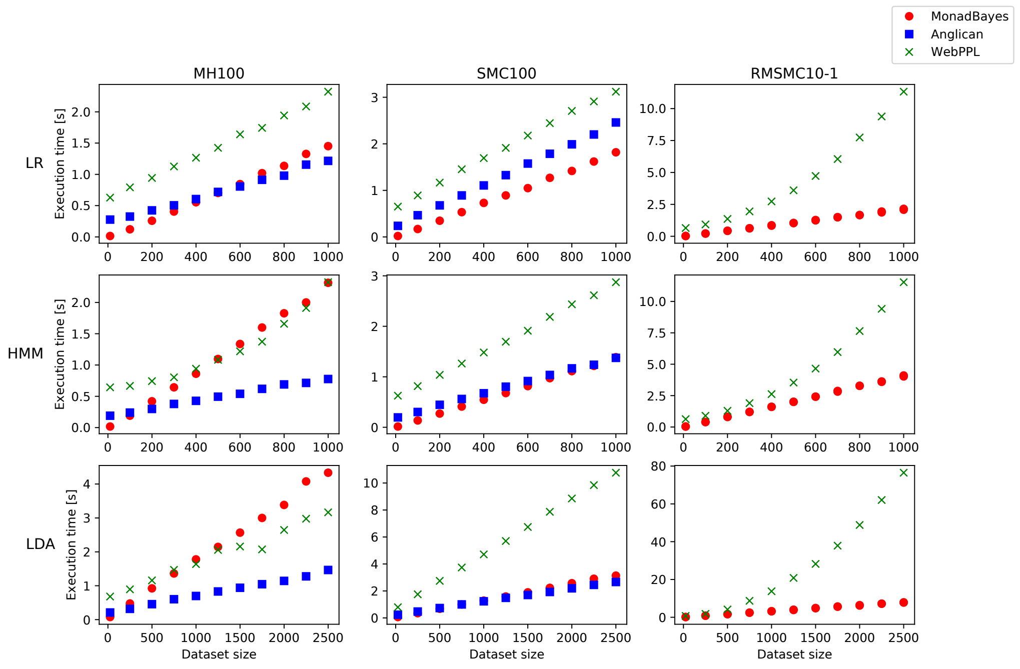

Fig. 3. Execution times of inference algorithms with varying dataset size. The numbers in the algorithm description indicate the parameters used. For MH we used 100 transitions, for SMC 100 particles, and for RM-SMC 10 particles and 1 rejuvenation step per particle per resampling step. The dataset size is the number of observations in LR and HMM and the total number of words in all documents in LDA.

Table 1. Net addition of Lines of Code (LoC), excluding comments and import statements. The entry “N/A” means that an algorithm is not available in a given language.

	MH	SMC	RM-SMC	PMMH	SMC^2
MonadBayes	67	70	11	4	20
Anglican	100	87	N/A	N/A	N/A
WebPPL	314	334	0	N/A	N/A

slope of the line associated with our library is significantly higher than for WebPPL. We speculate the cause to be that each `mhStep` in our library goes through the `Pop` layer while WebPPL only does it once per rejuvenation sequence. We leave reducing that overhead for future work.

5.2 Qualitative Evaluation

To estimate the implementation effort involved in writing the inference algorithms, Table 1 lists the number of lines of code (LoC) used for this purpose in the three systems. Although LoC are not a reliable metric, particularly comparing across languages, they do offer some estimation of the implementation effort. WebPPL requires no additional lines to implement RM-SMC because it implements SMC as a special case of RM-SMC with zero rejuvenation steps. The figure shows that our modular implementation of MH and SMC is actually shorter than monolithic implementations

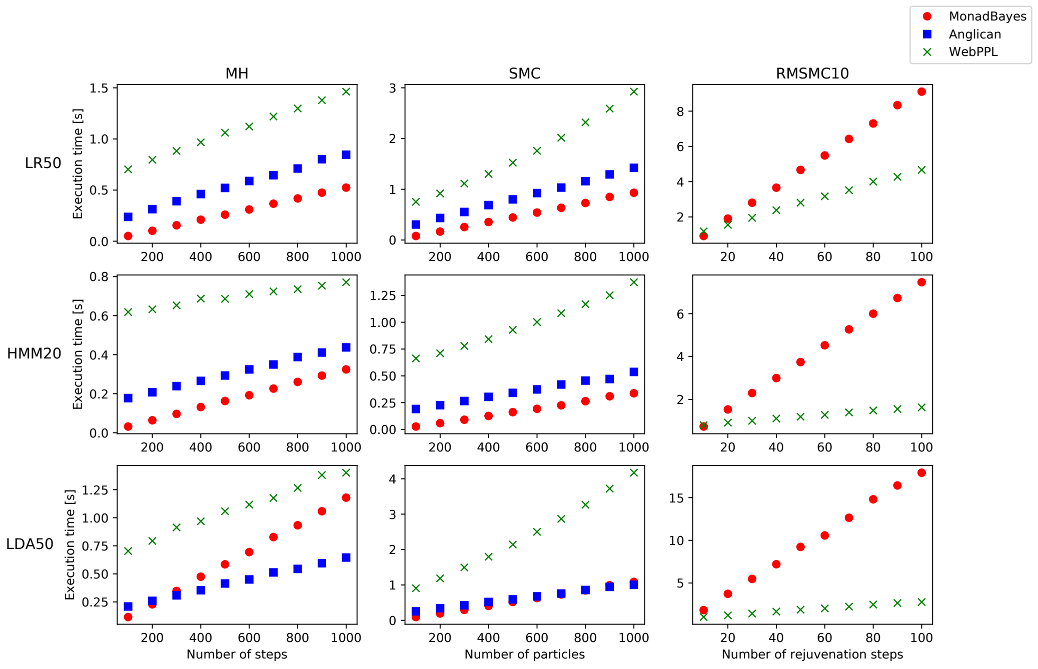

Fig. 4. Execution times of inference algorithms with varying sample size. The numbers after model names are the sizes of the datasets used. The number 10 in RMSMC10 indicates that we used 10 particles and only varied the number of rejuvenation steps. The X axis for RMSMC shows the number of rejuvenation steps applied per particle after each resampling operation.

in Anglican and WebPPL. Neither of these systems implements PMMH or SMC^2 which would involve substantial effort while they are short snippets of code in our library. We can expect this difference to become more and more pronounced as we continue to build complex inference algorithms from smaller and reusable building blocks.

Apart from the reduced number of LoC, our architecture makes the code more reliable, maintainable, and malleable for reasoning as it is close to Ścibior et al.'s semantic validation of inference 2018. In the next section, we also argue that our architecture enables modular testing. We expect these features to reduce the number of bugs in implementations and ease refactoring.

Finally, we profiled our implementation on these benchmarks to investigate the bottlenecks and suggest optimisations. The insights from profiling has lead us to two optimisations. First, we replaced `WriterT` with `StateT` in `W`, then we switched to a Church-encoded version of the free monad in `Tr`. We report the benchmark results after applying those optimisations. Profiling the final code reveals that a substantial amount of time, up to 90% depending on the benchmark, is spent on the overhead associated with inference representations and not on doing essential numeric work. The biggest offenders seem to be the `Pop` and `W` transformers. We hope that this overhead could be further reduced by suitable techniques but we do not see a clear way to do it. We include the profiler outputs in the supplementary material.

6 TESTING

An important benefit of the modular implementation is enhanced testing capabilities. Every inference transformation can be tested independently and the correctness of their composition follows from correctness of individual components. Furthermore, in certain circumstances it is possible to replace statistical tests with deterministic ones.

A standard approach to test an algorithm implementation is to compare with a reference implementation, less efficient but clearly correct, on a set of small examples. In the context of Bayesian inference we can use exact enumeration of a small discrete model like the sprinkler model from the introduction. Unfortunately, probabilistic algorithms only give approximate answers that can be arbitrarily bad with non-zero probability. We can therefore never be sure if the answers they provide are correct, and any statistical test is bound to produce both false positive and false negative results.

In our library we can perform deterministic tests of Monte Carlo methods by replacing the bottom monad `Sampler` with `Exact` that computes exact answers for discrete models. It is defined as follows, omitting conversions between `R` and `Log R`:

```
newtype Exact a = Exact {run :: [(a, Log R)]}
instance Monad Exact where
  return x = Exact [(x,1)]
  m >>= f = Exact
    [(y, p*q) | (x,p) ← run m, (y,q) ← run (f x)]
instance MonadSample Exact where
  random = error "Not␣available"
  bernoulli p = Exact [(True, p), (False, 1-p)]
instance MonadCond Exact where
  score w = Exact [((),w)]
normalForm :: Ord a ⇒ Exact a → [(a, Log R)]
-- sort, aggregate, and remove zeros
```

The function `normalForm` sorts the list according to the return values, the first components, aggregates weights of equal elements, and removes elements with zero weight. It allows us to compare distributions represented by lists for equality.

Any correct inference transformation should not alter the result of `Exact`. For example, if `~==` is an acceptable approximate floating-point equality, then we can write a deterministic test for `smc` as follows:

```
(normalForm . (>>= Exact) . runPopulation . smc 2 2)
  sprinkler ~== normalForm sprinkler
```

Our implementation of traces as `[R]` is fundamentally continuous so it does not work with `Exact`. However, a more elaborate trace type that distinguishes between continuous and discrete variables would enable us to write similar tests for `mhStep`.

Deterministic tests of the kind described above are limited in that they can only be applied to small discrete models and only verify certain aspects of correctness. In particular for SMC it only checks that the result is unbiased but not that it is consistent. Similarly a test for MH would only check that it preserves the posterior distribution but not that it converges to it. Nonetheless, we found those tests to be invaluable in practice. For example, if we forgot to preserve the total weight in `resample`, such a bug would quickly be caught by the test shown above.

7 OCAML IMPLEMENTATION

In the previous sections we present our design as a Haskell library, but the architecture is more generally applicable to languages with advanced functional programming features, in particular higher-order functions and inductive types. In statically typed languages we also require a sufficiently expressive type system that allows the high degree of polymorphism exploited in our design.

While the Haskell implementation is particularly elegant due to the availability of type-classes and higher-kinded polymorphism, these features are not strictly required to enjoy the benefits of modularity that our approach provides. To support this claim we sketch in this section an OCaml implementation that relies on the module system instead. This is currently a proof-of-concept work that shows how to port the key design patterns from the Haskell implementation, but we plan to eventually turn it into a complete probabilistic programming library.

We restrict ourselves to standard OCaml features although extensions such as modular implicits [White et al. 2015] would potentially make the implementation cleaner. Most of the Haskell features we used can be simulated with modules and functors, with the addition of custom record types to encode rank 2 types.

Module signatures replace the three type-classes we used in Haskell for inference representations, namely `Monad`, `MonadSample`, and `MonadInfer`. Since these type-classes extend one another we can simply include the relevant signatures.

```
module type Monad = sig
  type α t
  val return : α -> α t
  val (>>=) : α t -> (α -> β t) -> β t
  type nt  (* natural transformation *)
  val apply : nt -> α t -> α t
end
```

```
module type MonadSample = sig
  include Monad
  val random : float t
end

module type MonadInfer = sig
  include MonadSample
  val score : float -> unit t
end
```

The type `nt` is a custom record wrapping the type `α. α t -> α t`. We use it to represent inference transformations that maintain the representation type. This is necessary because only rank 2 functions that quantify over all `α` can be hoisted through inference transformers without breaking the abstractions.

We implement the inference representation `Sampler` as a concrete module. Since OCaml allows mutable state natively, the underlying type is just a thunk generating a value using a global random number generator. This type is exposed using the **`with`** annotation to enable the sampler to be executed by external code.

```
module Sampler : MonadSample with type α t = unit -> α =
struct
  type α t = unit -> α
  let return x = fun () -> x
  let (>>=) s f = fun () -> f (s()) ()
  type nt = {f : α. α t -> α t}
  let apply tau m = tau.f m
  let random = fun () -> Random.float 1.0
end
```

Inference transformers are implemented as functors over an abstract `MonadInfer` module. The functor contains a module representing an inference representation obtained by applying the transformer as well as any associated inference transformations. Inference transformations are represented as natural transformations when they do not modify the inference stack and as functions when they do. Note that this means only the former can be hoisted, although we could introduce an additional module type for the latter to make them rank 2 as well.

The population transformer is implemented as follows.

```
module type MonadPop = sig
  module Pop : MonadInfer
  type α m  (* transformed
              representation *)
  type h  (* natural transformation
           for m *)
  val lift : α m -> α Pop.t
  val hoist : h -> Pop.nt
  val run : α
Pop.t -> (α * float) list m
  val spawn : int -> Pop.nt
  val resample : Pop.nt
end
```

```
module Population (M : MonadSample) : MonadPop
  with type h = M.nt with type α m = α
M.t =
struct
  type α p = (α * float) list M.t
  type np = {f : α. α p -> α p}
  module Pop : MonadInfer
    with type α t = α p
    with type nt = np =
  struct
    (* ... *)
  end
  type α m = α M.t
  type h = M.nt
  (* ... *)
end
```

We can similarly implement the suspension transformer.

```
module type MonadSeq = sig
  module Seq : MonadInfer
  type α m
  type h
  val lift : α m -> α Seq.t
  val hoist : h -> Seq.nt
  val advance : Seq.nt
  val finish : α Seq.t -> α m
end
```

```
module Sequential (M : MonadInfer) : MonadSeq
  with type h = M.nt with type α m = α
M.t =
struct
  type (α, β) either = Left of α
| Right of β
  type α seq = Seq of (α, unit -> α
seq) either M.t
  type ns = {f : α. α seq -> α seq}
  type α t = α seq
  module Seq : MonadInfer
    with type α t = α seq
    with type nt = ns =
  struct
    (* ... *)
  end
  type α m = α M.t
  type h = M.nt
  (* ... *)
end
```

Compound inference transformations are implemented as separate modules. As a concrete example we implement SMC as a functor parameterized by the underlying inference representation. The `SMC` module contains two `MonadInfer` modules called `In` and `Out`, representing input and output inference representation types respectively. It also specifies a type for parameters to the algorithm and a function to apply the inference transformation. Other inference algorithms can be implemented as modules with the same type.

```
type smcparam = {steps : int; particles : int}

module SMC (M : MonadSample) : sig
  module In : MonadInfer
  module Out : MonadInfer
  type param
  val apply : param -> α In.t -> α Out.t
end
  with type param = smcparam
  with module Out = Population(M).Pop =
struct
  module P = Population(M)
  module S = Sequential(P.Pop)
  module In = S.Seq
  module Out = P.Pop
  type param = smcparam
  let rec applyN n f x =
    if n == 0 then
      x
    else
      applyN (n-1) f (f x)
  let apply {steps = k; particles = n} (model) =
    S.finish (applyN k (fun x -> S.Seq.apply S.advance
                              (S.Seq.apply (S.hoist P.resample) x))
                         (S.Seq.apply (S.hoist (P.spawn n)) model))
end
```

Finally, we implement models as functors parameterized over inference representations. The module signature is as follows.

```
module type Model = sig
  type α m
  type output
  val model : output m
end
```

The sprinkler example from the introduction would then be implemented as follows. Since OCaml does not have "do" syntax, we use >>= explicitly.

```
module Sprinkler (M : MonadInfer) : Model
  with type α m = α M.t with type output = bool =
struct
  type α m = α M.t
  type output = bool
  open M
  let bernoulli p = random >>= fun x -> return (x < p)
  let model =
    bernoulli 0.2 >>= fun rain ->
    bernoulli 0.1 >>= fun sprinkler ->
    let prob_lawn_wet =
      match (rain, sprinkler) with
        | (true , true ) -> 0.99
        | (true , false) -> 0.70
        | (false, true ) -> 0.90
        | (false, false) -> 0.01
    in
    score prob_lawn_wet >>= fun () ->
    return rain
end
```

To run the inference we simply need to instantiate the relevant modules.

```
module Alg = SMC(Sampler)
module Mod = Sprinkler(Alg.In)
module Out = Population(Sampler)
let sampler = Out.run (Alg.apply {steps = 1; particles = 3} Mod.model)
let results : (bool * float) list = sampler ()
```

8 RELATED WORK

The two approaches most related to ours are those of Ścibior et al. [2015] and Zinkov and Shan [2017]. Both of these compose inference algorithms as deterministic transformations of probabilistic programs. Ścibior et al. [2015] use an intermediate free monad representation that abstracts over deterministic parts of the program. While this representation allows them to compose inference transformations, using a fixed intermediate representation does not allow them the degree of flexibility equivalent to our transformers and as a result they can not implement algorithms such as RM-SMC, PMMH, or SMC^2. By constrast, Zinkov and Shan [2017] apply transformations directly to Hakaru source code. They target a different set of inference transformations than us, focusing on symbolic integration and designing custom MH kernels. Their approach is complementary to ours and we can envision applying their transformations to simplify the program before running one of our algorithms on it.

An alternative approach to composing inference algorithms is to explicitly construct a graph of random variables in the model and apply different algorithms to different regions of the graph. A standard example is the EM algorithm [Dempster et al. 1977], which can be implemented in a compositional manner as is done in systems such as Figaro [Pfeffer 2015] and Edward [Tran et al. 2017]. Other common uses of such compositions are custom MCMC algorithms combining different proposal kernels for different random variables, such as in PyMC [Patil et al. 2010], or message-passing algorithms that can be regarded as compositions of local inferences, such as in Infer.NET [Minka et al. 2014]. These types of compositions are also mostly orthogonal to the types we discuss in this paper.

Finally, we mentioned in Section 3.2 that more sophisticated representations of traces of probabilistic programs can improve efficiency of inference. In the context of MH the main benefit is the ability to better align random variables in different traces, as described by Wingate et al. [2011]. Ritchie et al. [2016] further describe how some of the computation involved is redundant and can be avoided. Mansinghka et al. [2014] retain the whole program structure in a trace and show how it can be used to incorporate custom extensions in the inference algorithms. All of these approaches can be used in conjunction with our implementation technique.

9 DISCUSSION AND FUTURE WORK

In this paper we presented a Haskell library for probabilistic programming that enables modular construction of inference algorithms through standard functional programming techniques.

The use of Haskell was convenient, as we could reuse the existing support for monads and their transformers for inference representations. However, the same design ports to other modern functional languages that contain higher-order functions and inductive types. In the absence of a type-class mechanism, one can use ML-style modules: users construct models abstractly with respect to a module signature containing `random` and `score`. Each inference representation is a module implementing this signature, and inference transformers are functors.

We have achieved performance comparable with existing probabilistic programming libraries available in general-purpose languages, although profiling shows there remains a significant overhead associated with the abstractions we use. While the convenience of a high-level language may in many cases already be worth reduced performance, additional work on minimising this overhead would make our approach and probabilistic programming in general even more practical.

Another source of improvement to the `Tr` representation consists of better trace representations. These include recording information flow dependencies, or limiting the variables that can be resampled. We hypothesise that many of the abstractions and representations from static analysis could lead to optimised representations. A better trace representation could also allow us to define more efficient MCMC kernels, either generically or on a per-model basis. Although we did not address this issue here, selecting good kernels is crucial for good performance of MCMC.

An alternative approach to embedding probabilistic programs in existing languages is to use a stand-alone DSL for constructing models, such as used in Stan [Carpenter et al. 2017] and LibBi [Murray 2013]. Such representations avoid the overheads discussed above and as a result can be used to generate computationally efficient inference code. It would be interesting to investigate if the approach we described here could be used to build modular compilers for such languages without sacrificing their efficiency. The idea to use monad transformers to build compilers in a modular fashion has been previously explored by Liang and Hudak [1996].

Many modern algorithms for Bayesian inference, such as Hamiltonian Monte Carlo [Neal 2010] and black-box variational inference [Ranganath et al. 2014] rely on gradient information that can be obtained by methods of automatic differentiation. Inclusion of these methods can make a probabilistic programming system dramatically more practical as demonstrated by the success of Stan. We could incorporate these algorithms into our framework, the only obstacle being availability of a suitable automatic differentiation library. We have experimented with the ad library in Haskell but the resulting types were so complicated that we did not find the results satisfactory, although it did work. We hope that a more clever implementation hiding type-level complexities or an alternative automatic differentiation package will unlock the power of gradient-based inference algorithms for our library.

Finally, modern machine learning, especially in the context of big data, increasingly makes use of dedicated numerical libraries for efficiently performing matrix operations, including automatic differentiation, on GPUs. Currently the most popular choices are Tensorflow [Abadi et al. 2015] and PyTorch[4]. Probabilistic programming systems such as Edward [Tran et al. 2017] and Pyro[5] exploit these libraries to deliver massive performance boosts. For example, Tran et al. [2017] report that Hamiltonian Monte Carlo in Edward is more than an order of magnitude faster than in Stan. Taking advantage of such libraries could therefore greatly improve computational efficiency of inference.

Currently, PyTorch is only available in Python and existing Tensorflow bindings for Haskell and OCaml are not supported officially. There are several alternatives written directly in functional programming languages that cover all the core features required although they presently do not easily combine. For example, the Haskell library Accelerate [Chakravarty et al. 2011] supports efficienct matrix computation on CPUs and GPUs, but there is no automatic differentiation library built on top of it. On the other hand a recently released Owl library [Wang 2017] for OCaml offers efficient matrix operations and automatic differentiation, but currently only emits CPU code. Since all the components are already there, it is simply a matter of investing sufficient developer time to create a natively functional numerical library for modern machine learning. We believe that in the near future such libraries, whether implemented from scratch in functional languages or as convenient bindings to imperative libraries, will make functional programming a serious player in the area of machine learning in general and probabilistic programming in particular. We hope that the constructions presented in this paper will help bring the modularity and reliability often associated with functional programming into the realm of state-of-the-art machine learning applications.

ACKNOWLEDGMENTS

Supported by the Engineering and Physical Sciences Research Council grant EP/N007387/1 'Quantum computation as a programming language', and a Balliol College Oxford Career Development Fellowship. We thank David Tolpin for providing Anglican and WebPPL scripts we used for benchmarking and Jeremy Yallop for his help with using the OCaml type system. We also thank Rob Zinkov and the anonymous reviewers for providing us valuable feedback.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. 2011. Noncomputable Conditional Distributions. In *LiCS*. http://ieeexplore.ieee.org/document/5970208/

Chrisophe Andrieu, Arnaud Doucet, and Roman Holenstein. 2010. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society* 72 (2010), 269–342. www.stats.ox.ac.uk/~doucet/andrieu_doucet_holenstein_PMCMC.pdf

David Barber. 2012. *Bayesian Reasoning and Machine Learning*. Cambridge University Press. https://doi.org/10.1017/CBO9780511804779

Christopher Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer-Verlag New York. http://www.springer.com/gb/book/9780387310732

Mario Blažević. 2011. Coroutine Pipelines. *The Monad Reader* (2011), 29–50. Issue 19.

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

[4]https://github.com/pytorch
[5]http://pyro.ai/

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

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *DAMP*.

Nicolas Chopin, Pierre E. Jacob, and Omiros Papaspiliopoulos. 2013. SMC2: an efficient algorithm for sequential analysis of state space models. *Journal of the Royal Statistical Society Series B: Statistical Methodology* 75 (2013), 397–426. Issue 3. http://onlinelibrary.wiley.com/doi/10.1111/j.1467-9868.2012.01046.x/abstract

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 315–326. https://doi.org/10.1145/1291151.1291199

Arthur P. Dempster, Nan M. Laird, and Donald B. Rubin. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society Series B: Statistical Methodology* 39 (1977), 1–38. Issue 1. https://mathscinet.ams.org/mathscinet-getitem?mr=0501537

Arnaud Doucet and Adam M. Johansen. 2011. A Tutorial on Particle Filtering and Smoothing: Fifteen years later. In *The Oxford Handbook of Nonlinear Filtering*, Dan Crisan and Boris Rozovskii (Eds.). Oxford University Press, Chapter 8.

Walter Gilks and Carlo Berzuini. 2001. Following a moving target - Monte Carlo inference for dynamic Bayesian models. *Journal of the Royal Statistical Society* 63 (2001), 127–146. www.mathcs.emory.edu/~whalen/Papers/BNs/MonteCarlo-DBNs.pdf

W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. 1994. A Language and Program for Complex Bayesian Modelling. *Journal of the Royal Statistical Society. Series D* 43 (1994).

Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua Tenenbaum. 2008. Church: a language for generative models. In *UAI*. http://cocolab.stanford.edu/papers/GoodmanEtAl2008-UncertaintyInArtificialIntelligence.pdf

Noah Goodman and Andreas Stuhlmüller. 2014. Design and Implementation of Probabilistic Programming Languages. http://dippl.org. (2014).

Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *LiCS*. http://ieeexplore.ieee.org/document/8005137/

Sheng Liang and Paul Hudak. 1996. Modular Denotational Semantics for Compiler Construction. In *ESOP*.

David J. C. MacKay. 2003. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press. www.inference.org.uk/itila/book.html

Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. arXiv:1404.0099. (2014).

Tom Minka, John Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. 2014. Infer.NET 2.6. Microsoft Research Cambridge. http://research.microsoft.com/infernet. (2014).

Lawrence M. Murray. 2013. Bayesian State-Space Modelling on High-Performance Hardware Using LibBi. arXiv:1306.3277. (2013).

Radford M. Neal. 2010. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*. Chapman and Hall.

Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. 2014. R2: An Efficient MCMC Sampler for Probabilistic Programs. In *AAAI*.

Anand Patil, David Huard, and Christopher J. Fonnesbeck. 2010. PyMC: Bayesian Stochastic Modelling in Python. *Journal of Statistical Software* 35 (2010).

Avi Pfeffer. 2001. IBAL: A Probabilistic Rational Programming Language. In *IJCAI*.

Avi Pfeffer. 2015. *Practical Probabilistic Programming*. Manning. https://www.manning.com/books/practical-probabilistic-programming

Rajesh Ranganath, Sean Gerrish, and David Blei. 2014. Black-Box Variational Inference. In *AISTATS*. http://www.jmlr.org/proceedings/papers/v33/ranganath14.html

Daniel Ritchie, Andreas Stuhlmüller, and Noah D. Goodman. 2016. C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching. In *AISTATS*. http://proceedings.mlr.press/v51/ritchie16.html

Adam Ścibior, Zoubin Ghahramani, and Andrew Gordon. 2015. Practical Probabilistic Programming with Monads. In *Haskell*. http://dl.acm.org/citation.cfm?id=2804317

Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K. Moss, Chris Heunen, and Zoubin Ghahramani. 2018. Denotational Validation of Higher-Order Bayesian Inference. *Proceedings of the ACM on Programming Languages* 2 (2018).

Sam Staton. 2017. Commutative semantics for probabilistic programming. In *Proc. ESOP 2017*.

David Tolpin, Jan-Willem van de Meent, and Frank Wood. 2015. Probabilistic Programming in Anglican. In *Machine Learning and Knowledge Discovery in Databases*, Albert Bifet, Michael May, Bianca Zadrozny, Ricard Gavalda, Dino Pedreschi, Francesco Bonchi, Jaime Cardoso, and Myra Spiliopoulou (Eds.). Lecture Notes in Computer Science, Vol. 9286. Springer International Publishing, 308–311. https://doi.org/10.1007/978-3-319-23461-8_36

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep Probabilistic Programming. In *ICLR*.

Liang Wang. 2017. Owl: A General-Purpose Numerical Library in OCaml. arXiv:1707.09616. (2017).

Leo White, Frédéric Bour, and Jeremy Yallop. 2015. Modular Implicits. ACM Workshop on ML 2014 post-proceedings. (September 2015).

David Wingate, Andreas Stuhlmüller, and Noah Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *AISTATS*. https://web.stanford.edu/~ngoodman/papers/lightweight-mcmc-aistats2011.pdf The published version contains a serious bug in the algorithm description, which was fixed in Revision 3 available from the authors page.

Frank Wood, Jan-Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *AISTATS*. http://www.robots.ox.ac.uk/~fwood/assets/pdf/Wood-AISTATS-2014.pdf

Robert Zinkov and Chung-chieh Shan. 2017. Composing inference algorithms as program transformations. In *UAI*.