Practical Probabilistic Programming with Monads

Adam Ścibior  
University of Cambridge  
ams240@cam.ac.uk

Zoubin Ghahramani  
University of Cambridge  
zoubin@eng.cam.ac.uk

Andrew D. Gordon  
Microsoft Research  
and University of Edinburgh  
adg@microsoft.com

Abstract

The machine learning community has recently shown a lot of interest in practical probabilistic programming systems that target the problem of Bayesian inference. Such systems come in different forms, but they all express probabilistic models as computational processes using syntax resembling programming languages. In the functional programming community monads are known to offer a convenient and elegant abstraction for programming with probability distributions, but their use is often limited to very simple inference problems. We show that it is possible to use the monad abstraction for constructing probabilistic models, while still offering good performance of inference in challenging models. We use a GADT as an underlying representation of a probability distribution, and apply Sequential Monte Carlo-based methods to achieve efficient inference. We define a formal semantics via measure theory and check the monad laws. We demonstrate a clean and elegant implementation that achieves performance comparable with Anglican, a state-of-the-art probabilistic programming system.

1. Introduction

1.1 Bayesian Models for Machine Learning

The paradigm of Bayesian modelling is to express one’s beliefs about the world as a probability distribution over some suitable space, and then to use observed data to update this distribution via Bayes’ rule

\[ P(\theta|D) = \frac{1}{Z} P(\theta) P(D|\theta) \]

where \(\theta\) denotes the model parameters (the unobservable beliefs about the world), \(D\) is the observed data, \(P(\theta)\) is the prior distribution over the parameters (beliefs before the data is observed), \(P(\theta|D)\) is the posterior distribution (beliefs after data is observed), \(P(D|\theta)\) is the likelihood (probability of generating observed data given particular values of parameters), and \(Z\) is a normalising constant that ensures the posterior is a proper probability distribution.

For example, in a model for linear regression, the data \(D\) consists of a set of points \((x_i, y_i)\), while the parameters \(\theta = (A, B)\) are the slope \(A\) and intercept \(B\) of the best-fitting line \(y = Ax + B\) through the points.

Bayesian inference is the question of computing the posterior \(P(\theta|D)\), so as to make predictions or decisions. In the example of linear regression, inference consists of computing the posterior distribution of the slope and intercept, \(P(A, B | (x_i, y_i))\), which may be used to predict \(y'\) given an unseen \(x'\), for instance.

Although Bayesian modelling is a robust and flexible framework, inference is often intractable. This problem is usually addressed in several ways. First of all, the prior and the likelihood are often chosen to be simple distributions, computationally tractable if not having analytical closed forms. Sometimes this is enough to compute the posterior exactly. If not, a wide range of approximate inference methods can be used to approximately compute the posterior. We will not review here the vast field of approximate inference, but instead we refer to Barber (2012). We are primarily concerned with Monte Carlo methods for approximate inference.

1.2 Probabilistic Programming for Bayesian Inference

In traditional approaches to Bayesian models, the mechanics of inference is tightly coupled with the model description. Instead, in probabilistic programming (Goodman 2013; Gordon et al. 2014), the intent is to make models and inference algorithms more composable and reusable by specifying the model as a piece of probabilistic code, and implementing Bayesian inference as an interpreter or compiler for the probabilistic code.

There are many probabilistic programming languages with different trade-offs. Many languages, such as BUGS (Gilks et al. 1994), Infer.NET (Minka et al. 2009), and Stan (Stan Development Team 2014), restrict the underlying deterministic language by disallowing recursion, to ensure that the model can be compiled to a finite graphical model. This approach simplifies writing inference algorithms, but at the price of reduced expressiveness. Others such as HANSEI (Kiselyov and Shan 2009) use sophisticated interpretation techniques, but are limited to discrete distributions.

Many different solutions were proposed to extend probabilistic programming systems to potentially infinite models, such as combining graphical models with first-order logic, as in BLOG (Milch et al. 2005), Markov Logic Networks (Domingos and Richardson 2004), and ProbLog (Kimmig et al. 2011).

The most general approach, known as universal probabilistic programming, is to use a Turing-complete language. The pioneer is Church (Goodman et al. 2008), a functional subset of Scheme equipped with primitives for constructing standard probability distributions and a conditioning predicate. Other universal probabilistic programming languages include Venture (Mansinghka et al. 2014) and Anglican (Wood et al. 2014), both essentially dialects of Scheme. These universal languages are more expressive than those that compile to factor graphs. For example, they support nonparametric models such as the Dirichlet Process, useful for learning the best number of clusters from the data, but which do not compile to finite factor graphs. Inference for these more expressive models is
1.3 Universal Probabilistic Programming in Haskell

The purpose of this paper is to provide evidence that the general-purpose functional language Haskell is an excellent alternative to special-purpose languages like Church, for authoring Bayesian models and developing inference algorithms.

We embrace the idea of writing Bayesian models using the probability monad (Ramsey and Pfeffer 2002; Erwig and Kollmansberger 2006; Larsen 2011). We go beyond by developing a series of practical inference methods for monadic models, as opposed to relying on precise but unscaleable methods of exhaustive enumeration of discrete distributions. We show Haskell code for a range of rich models, including the Dirichlet Process, of the sort otherwise written in Church and its relatives. We demonstrate that state-of-the-art Monte Carlo inference algorithms, including Sequential Monte Carlo (Doucet et al. 2000), Particle Independent Metropolis Hastings (Andrieu et al. 2010), and Particle Cascade (Paige et al. 2014), written in Haskell, have performance better than the special-purpose Anglican implementation (Wood et al. 2014), and comparable to Probabilistic C (Paige and Wood 2014).

As a formal foundation for reasoning about models and inference, we present a formal semantics for our monad. The semantics is in two stages, inspired by recent work on the semantics of probabilistic while-programs (Hur et al. 2015). In the first stage, we interpret the monad as a deterministic function that consumes a stream of random samples to produce results. In the second stage, we define the distribution (a probability measure) given by closed expression of monadic type, by integrating its results over arbitrary random tapes.

To the best of our knowledge, this is the first rigorous semantics for a higher-order probabilistic programming language with conditioning (such as Church, Venture, or Anglican).

The main contributions of the paper are as follows.

- A practical library for probabilistic programming with monads. Previous approaches were elegant but inefficient. Here we implement state-of-the-art inference algorithms and performance is comparable with Anglican implementation.
- Randomised inference algorithms are viewed as deterministic transformations of a data structure representing a distribution. This makes inference clean, easy to implement, and sometimes even compositional.
- We demonstrate that a lazy probabilistic programming language allows for easy implementation of the Dirichlet Process model and the Particle Cascade algorithm.
- We present a formal semantics based on measure theory.

1.4 Structure of the Paper

We start in section 2 by showing on multiple examples how to build probabilistic models with monads. We specify what additional interface functions are required for a probabilistic programming system for Bayesian inference and we demonstrate how to use them. In section 3 we present an implementation of a GADT that satisfies those requirements and we explain how the inference problem manifests itself in this context. Then in section 4 we discuss implementation of several sampling-based methods for performing inference in probabilistic programs. In section 5 we present empirical evidence to correctness of our implementation and compare its performance with a state-of-the-art probabilistic programming system Anglican, which implements the same set of inference methods. We also define formal semantics of our approach to probabilistic programming in section 6 and we use it to prove the monad laws.

Finally, in section 7 we discuss related work and in section 8 we conclude.

1.5 Intended audience

Even though this paper is mainly directed towards the functional programming audience, we hope that it will also be of interest to the machine learning community. For this reason we include in the paper some material that may be difficult to read for people without deep knowledge of Bayesian statistics. We took care to build up slowly from simple models and algorithms, where possible appealing to intuition. In particular sections 2 and 4 start with simple examples, but end with sophisticated models and algorithms which require substantial background in Bayesian methods to understand. It is our hope that even readers without such background will be able to understand the main points of this paper.

2. Modelling with monads

In this section we define and motivate the requirements we place on an implementation of probability distributions. Those requirements result in an interface, which we use to define probabilistic models. A large part of this section consists of examples that demonstrate how to use this interface.

2.1 Requirements

We introduce a type Prob to represent probabilities and probability densities. It is implemented as a Double, but sometimes it is useful to distinguish between the two. Another advantage is that we could easily replace Double with LogFloat to improve numerical stability.

```
newtype Prob = Prob {toDouble :: Double}
  deriving (Show, Eq, Ord, Num, Fractional, Real, RealFrac, Floating, Random, Ext.Distribution Ext .StdUniform)
prob :: Double -> Prob
```

We represent probability distributions over type a using a datatype Dist a. We place the following requirements on Dist:

1. availability of standard distributions to be used as building blocks, such as

```
uniform :: [a] -> Dist a
categorical :: [(a,Prob)] -> Dist a
normal :: Double -> Double -> Dist Double
beta :: Double -> Double -> Dist Double
gamma :: Double -> Double -> Dist Double
```

2. Monad instance to combine distributions into sophisticated models

```
instance Monad Dist where
```

3. conditioning function, which takes a likelihood and a prior, and can be used to observe some data generated by the model

```
condition :: (a -> Prob) -> Dist a a -> Dist a
```

4. a way to sample from a Dist

```
sample :: StdGen -> Dist a a -> a
```

where StdGen is a random number generator.

Together those four requirements form an interface for working with Dist. In the remainder of this section we show how to use this interface to build probabilistic models of increasing complexity.
2.2 Dice rolling

We start with a very simple example. Suppose we roll \( n \) six-sided dice and look at the sum of the values in top faces. A distribution over such outcomes can be written in the following way:

\[
die :: \text{Int} \to \text{Dist} \text{Int}
die 0 = \text{return} 0
die 1 = \text{uniform} [1..6]
die n = \text{liftM2} (+) (\text{die} 1) (\text{die} (n-1))
\]

We can simulate one roll using the \text{sample} function and we can use standard functions \text{replicate} and \text{sequence} to generate a distribution over a number of independent rolls.

\[
\text{result} = \text{sample} \ g \ (\text{die} \ n)
\text{results} = \text{sample} \ g \ \text{sequence} \ \text{repeat} \ k \ \text{die} \ n
\]

An example run would be

\[
> \text{let} \ (n,k) = (3,5) > \text{let} \ g = \text{mkStdGen} 0 > \text{sample} \ g \ \text{die} \ n
\]
\[
14 > \text{sample} \ g \ \text{sequence} \ \text{repeat} \ k \ \text{die} \ n
[10,8,16,12,10]
\]

This style of constructing a list of independent samples from a distribution turns out to be very useful and we use it extensively in this paper. It is worth noting here that our implementation of \text{Dist} is lazy, so it is equivalent to write

\[
\text{results} = \text{take} \ k \ \text{sample} \ g \ \text{sequence} \ \text{repeat} \ \text{die} \ n
\]

It would be natural to attempt different operations on the distribution, for example we could ask what is the probability of a specific outcome or outcomes that satisfy a certain condition. However, in this paper we are only concerned with sampling. The reason for that is we are primarily concerned with solving difficult problems where exact answers are intractable and we need to resort to approximate sampling. With a collection of samples we can approximately answer any query about a distribution.

2.3 Coin tossing

We now turn to another toy problem which demonstrates how to use conditioning. We are given a coin, which may be fair or biased, and we toss it several times to determine which is the case. For the coin landing \( w \) with probability \( \text{isFair} \) we define a likelihood function.

\[
\text{toss} :: \text{Bool} \to \text{Dist} \text{Prob} 
\text{toss} \ b = \text{condition} (w \to \text{if} \ b \ \text{then} \ w \ \text{else} \ 1 - w)
\]

The posterior over the weight of the coin is:

\[
\text{posteriorWeight} = \text{tosses} \ \text{observations} \ \text{weight}
\]

In order to get any useful results from the posterior we actually need to perform inference. We delay the discussion of how to do it until section 4.

2.4 Linear regression

A very simple but very useful statistical model is that of linear regression, where we try to find a straight line that fits a set of observed data points best. The construction of this model is very similar to the coin tossing example above, so we only offer a brief explanation here. We put independent Gaussian priors over the slope and the intercept and we assume the noise is Gaussian too.

\[
\text{linear} :: \text{Dist} \text{Prob} 
\text{linear} = \text{do} \ a \leftarrow \text{normal} 0 1 \ b \leftarrow \text{normal} 0 1 \ \text{return} \ (a,b)
\]

In the above \text{Normal} is a data constructor for a datatype representing a normal distribution. It is available in several libraries and we do not define it here, but we emphasise it is not the same as \text{Normal}, which constructs a \text{Dist}. The \text{pdf} function computes the probability density function of a given distribution at a given point.

The above examples are very simple and do not demonstrate the power of the monadic approach to probabilistic modelling. We now demonstrate several standard probabilistic models expressed in a monadic form.

2.5 Hidden Markov Model

A Hidden Markov Model (HMM) is very popular for modelling sequential data. It consists of two sequences of random variables. The latent sequence is assumed to be Markovian and stationary, that is each of the latent variables only depends on the value of the previous one and the form of the dependency is the same for all elements. In the observed sequence each random variable only depends on the value of the associated latent variable, again in a stationary way. We aim to infer the values of the latent variables given the observed ones. In this particular model we assume that latent variables are discrete, while the observable ones are continuous.

\[
\text{hmm} :: \text{Dist} \text{Int} 
\text{hmm} = \text{liftMreverse} \ \text{states} \ \text{where} 
\text{states} = \text{foldl} \ \text{expand} \ \text{start} \ \text{values} 
\text{expand} = \text{foldl} \ \text{expand} \ \text{start} \ \text{states} 
\text{expand} d y = \text{condition} (\text{score} \ y \ . \ \text{head}) \ \text{do} \ \text{rest} \leftarrow d \ x \leftarrow \text{trans} \ \text{head} \ \text{rest} \ \text{return} (x:\text{rest}) 
\text{score} y x = \text{pdf} (\text{Normal} x 1) y 
\text{trans} 0 = \text{categorical} \ \text{zip} [-1..1] [0.1,0.4,0.5] 
\text{trans} 1 = \text{categorical} \ \text{zip} [-1..1] [0.2,0.6,0.2] 
\text{trans} 2 = \text{categorical} \ \text{zip} [-1..1] [0.15,0.7,0.15] 
\text{start} = \text{uniform} [-1..1]
\]
values = [0.9, 0.8, 0.7, 0.025, 5, 2, 0.1, 0, 0.13, 0.45, 6, 0.2, 0.1, -1, -1]

This particular way of implementing an HMM may be somewhat confusing. After all, we could simply expand the entire sequence of latent states and then use it to generate likelihoods of the observations. However, to make inference efficient, it is important to generate likelihoods as early as possible and to use them only on the left side of the monadic bind. We will discuss those points further in section 4.

2.6 Dirichlet Process mixture of Gaussians
A common task in machine learning is clustering. The problem is, given a set of data points, to separate it into disjoint clusters based on similarity. For continuous data points a common model choice is a mixture of Gaussians, where the likelihood of a point belonging to a particular cluster is given by a normal distribution. There are, however, many possible choices for deciding on the number of clusters. Sometimes it is possible to fix the number of clusters in advance, sometimes it is necessary to put a prior distribution over the number of clusters. We choose to use a Dirichlet Process (DP), where the number of clusters is not determined by a specific parameter, but rather can grow with the number of data points. The complete review of the DP is beyond the scope of this paper and we encourage interested readers to consult the specialised sources (Teh 2010). We merely note that we are using a stick-breaking representation here, where a DP is generated using an infinite sequence of random variables. As our implementation of Dist is lazy, we can code this representation directly.

```
pick :: [Prob] -> [a] -> Dist a
pick (b:breaks) (a:atoms) = do
  keep <- bernoulli b
  if keep then return a else pick breaks atoms
```

dpMixture :: Dist [Int]
dpMixture =
  let
    -- lazily generate clusters
    clusters = do
      let atoms = [1..]
      breaks <- sequence $ repeat $ fmap prob $ beta 1 1
      let classgen = stick breaks atoms
      vars <- sequence $ repeat $ fmap (1/) $ gamma 1 1
      means <- mapM (normal 0) vars
      obs = [1.0, 1.1, 1.2, -1.0, -1.5, -2.0, 0.001, 0.01, 0.005, 0.0]
    n = length obs
    -- start with no data points
    start = fmap ([],[]) clusters
    -- add points one by one
    points = foldl build start obs
    build d y = condition (score y . head . snd) $ do
      (clusters, rest) <- d
      let (classgen, vars, means) = clusters
      cluster <- classgen
      let point = (cluster, vars !! cluster, means !! cluster)
      return (clusters, point : rest)
    -- the likelihood in a cluster in Gaussian
    score y (cluster, var, mean) =
      -- Normal mean stddev
      prob $ pdf (Normal mean (sqrt var)) y
  in
  -- extract cluster assignments
  fmap (reverse . map (\(x,_,_) -> x) . snd) points

3. Implementation
In the previous section we showed how to use the monadic interface to build probabilistic models. Here we discuss an implementation of the underlying data structure Dist, focusing in particular on the task of performing inference.

3.1 List of values
The simplest possible implementation would be a list of weighted values, such as suggested by Erwig and Kollmansberger (2006). This approach is very easy to understand, but it is also very limited. It can not be used with continuous distributions and it is tied to a particular, inefficient inference strategy. Nonetheless, we present this implementation below, in the hope that it makes the semantics of the interface easier to understand. We call this implementation Explicit, since it represents a distribution explicitly as a collection of weighted values.

```
newtype Explicit a = Explicit {toList :: [(a,Prob)]}
instance Functor Explicit where
  fmap f (Explicit xs) = Explicit $ map (first f) xs
instance Monad Explicit where
  return x = Explicit [(x, 1)]
  (Explicit xs) >>= f =
    Explicit [(y,p*q)| (x,p) <- xs, (y,q) <- toList (f x)]
  condition (Explicit xs) c = Explicit $ normalize $ reweight c xs
  reweight c xs = map (\(x,p) -> (x, p * c x)) xs
  normalize xs = map (second (/ w)) xs where
    w = sum $ map snd xs
  sample g (Explicit xs) =
    scan r xs where
      r = length obs
    -- start with no data points
    start = fmap ([],[]) clusters
    -- add points one by one
    points = foldl build start obs
    build d y = condition (score y . head . snd) $ do
      (clusters, rest) <- d
      let (classgen, vars, means) = clusters
      cluster <- classgen
      let point = (cluster, vars !! cluster, means !! cluster)
      return (clusters, point : rest)
    -- the likelihood in a cluster in Gaussian
    score y (cluster, var, mean) =
      -- Normal mean stddev
      prob $ pdf (Normal mean (sqrt var)) y
```

3.2 GADT
There is no single best inference method to be used in all cases, so we decide on an implementation that provides as much flexibility as possible. Specifically, we implement Dist as a GADT which simply stores the applications of all the functions exposed by the interface.

```
data Dist a where
  Return :: a -> Dist a
  Bind :: Dist b -> (b -> Dist a) -> Dist a
  Primitive :: Dist Double
  Conditional :: (a -> Prob) -> Dist a -> Dist a
condition = Conditional
instance Functor Dist where
  fmap = liftM
instance Monad Dist where
  return = Return
  (>>=) = Bind
```

```
data Dist a where
  Return :: a -> Dist a
  Bind :: Dist b -> (b -> Dist a) -> Dist a
  Primitive :: Dist Double
  Conditional :: (a -> Prob) -> Dist a -> Dist a
condition = Conditional
instance Functor Dist where
  fmap = liftM
instance Monad Dist where
  return = Return
  (>>=) = Bind
```

4 2016/2/28
Here we only use Primitive, a uniform distribution on the \([0, 1]\) interval, as a basic building block. All other standard distributions can be built from one or more uniform random variables using monadic operations. We only do this for clarity and in practice we allow an arbitrary sampler to be used as a basic building block. In particular all the standard distributions can be pulled directly from existing libraries with very little effort.

A Dist can only be sampled from if it is not conditioned. This design choice is motivated by the fact that conditioning is a declarative description of the posterior, which does not specify how to sample from it. Before a conditional distribution can be sampled from, we need to apply an inference algorithm that specifies a way of sampling from the posterior, either exactly or approximately. For this reason we choose to view inference as a deterministic Dist to Dist transformation. An inference method, several of which are discussed in section 4, converts a Dist into an equivalent Dist, but one without conditionals. The notion of equivalence of distributions is formally defined in section 6. In practice we can often get better results by trying to approximate a Dist with a collection of samples. For this reason we also consider inference methods that transform a Dist into Dist \([a]\) or Dist \(((a,\text{Prob}))\). In principle we could recover the relevant Dist \(a\) by sampling from such a collection at the end. We show some examples of this later on.

4. Inference

The central question in probabilistic modelling is how to do inference efficiently. In the previous section we explained that we interpret inference as a Dist to Dist transformation that preserves semantics while removing conditionals. Ideally inference methods would just be functions \(\text{Dist}a \rightarrow \text{Dist}a\), but in practice this would often discard useful information. For this reason we provide two implementations of every inference algorithm. An inference function produces a distribution over collections of samples, while inference’ produces a distribution over single values.

We present implementations of several inference methods, in particular of the Particle Markov Chain Monte Carlo (PMCMC) (Andrieu et al. 2010) methods that were first used for probabilistic programming in Anglican (Wood et al. 2014). Some of them require the presence of Conditionals to be independent of any random choices in the model. In modelling terms this is a reasonable assumption, since the observed data is fixed. To enforce this constraint we require that all the Conditionals should be placed on the left of the monadic Bind. We could use the type system to enforce this constraint, but then we could no longer make Dist an instance of the Monad class.

4.1 Sampling from the prior

The very first thing we show is how to draw samples from the prior. The prior can be regarded as the simplest approximation to the posterior, but not one relevant in practice. We still discuss it here for completeness. Algorithmically sampling from the prior amounts to discarding Conditionals in a Dist. As discussed above, the Conditionals should only be placed on the left of Bind, so we do not need to worry about what is right of Bind.

4.2 Importance sampling

The subsequent inference methods rely on sampling from the prior, but rather than discarding the likelihood score they retain it and use it to reweight the samples. For this purpose we define a function that accumulates the scores rather than discarding them.

4.3 Metropolis-Hastings

Another way to obtain the correct posterior by drawing samples from a different proposal distribution is Markov Chain Monte Carlo (MCMC) (Neal 1993). Perhaps the most popular MCMC method used for Bayesian inference is the Metropolis-Hastings (MH) algorithm. The MH algorithm generates an infinite sequence of samples, called a Markov Chain, by proposing a new sample from a proposal distribution and accepting or rejecting it with probability proportional to the ratio of scores between the current sample and the proposed one. In case of rejection the current sample is retained as the next sample. It can be proven that under certain mild conditions the marginal distributions of subsequent samples converge to the true posterior, regardless of the starting point. In practice, however, we often take the whole sequence of samples to approximate the posterior.

Here we use a very simple variant of MH where the proposal distribution is the prior and the score is the likelihood. For the readers familiar with the existing probabilistic programming literature, note that this is not a single-site MH as proposed by Wingate et al. (2011), but rather an MH that proposes the entire trace from the prior.
Conditional

that is all

the presence

Conditional

with

smc

information and recover the ordinary SMC algorithm by composing

obtain better results. In particular, it is possible to discard the extra

used to correct for bias introduced by SMC. The reason we do it

It is useful to retain the pseudo-marginal likelihood, since it can be

new scores are weighted averages of likelihood scores across all the

smc

hoods. We can accomplish it by composing

by weighting the particles according to the pseudo-marginal likeli-

4.4 Sequential Monte Carlo

There exists a more powerful inference method, somewhat similar
to importance sampling, called Sequential Monte Carlo (SMC)
(Doucet et al. 2000). Originally SMC was used only with sequential
data, where latent variables can be arranged in a sequence with
some data being observed at each step. SMC approximates the
partial posterior at each step by a collection of weighted samples,
called particles, which are reweighted according to the likelihood
of the observed data at each step. In order to avoid accumulating
excessive weight on a small number of particles, the particles are
resampled at each step.

It was recently shown that SMC can be applied to probabilistic
programs (Wood et al. 2014) and similarly we can apply it to do
inference on a Dist. The essential feature that allows for it is that
the presence Conditionals is not affected by any random choices,
that is all Conditionals are to the left of Bind.

4.5 Particle Independent Metropolis Hastings

The idea that SMC can be used as a part of a more powerful
inference algorithm in not new. In particular, there exist a family
of MCMC methods, called Particle MCMC (PMCMC) (Andrieu
et al. 2010), which use SMC to obtain a proposal distribution for the
MH algorithm. Perhaps the simplest PMCMC algorithm is Particle
Independent Metropolis-Hastings (PIMH), which is an MH with
proposals generated by independent SMC runs and the scores equal
to pseudo-marginal likelihoods. We can obtain PIMH simply by
composing smc with mh.

4.6 Particle Cascade

A final inference algorithm that we discuss here is the recently pro-
posed Particle Cascade (PC) (Paige et al. 2014). PC is essentially
SMC with an infinite number of particles, where resampling is done
only based on previous particles and not on subsequent ones. The
result of PC is an infinite sequence of samples that can be consumed
lazily until desired accuracy is achieved. Within a lazy probabilistic
programming language we can implement PC in almost the same
way as SMC, changing only the resampling function.

5. Evaluation

We evaluate our implementation by running some of the inference
algorithms from section 4 on selected models from section 2. Our
purpose is to demonstrate correctness and efficiency of implement-
ation, not to investigate relative performance of different inference
methods. In order to establish correctness we compare the empir-
ical posterior consisting of samples obtained from inference algo-
rithms to an exact posterior. We carefully select models for which
we can compute the exact posterior by a combination of analyti-
cal and computational methods. We use KL divergence, a standard
metric of dissimilarity between distributions, to compare our results
to the exact posteriors. For a correct sampler the KL divergence
should decay approximately according to a power law, producing a straight line on a log-log scale. The results are given in figure 1.

We judge efficiency of our implementation by comparing execution times with Anglican and Probabilistic C. We run all the experiments using a single core of an Intel Core i7 CPU 920 @ 2.67GHz on a machine running Ubuntu 14.04. Although Anglican and Probabilistic C can run inference utilising multiple cores, we only used one to get a better comparison with our implementation, which is currently sequential. In section 8 we briefly discuss how it could be parallelised. Anglican was originally an interpreted language, but was recently reimplemented \(^1\) by compiling to Clojure code directly. We compare our implementation against the new, faster version (0.6.5).

We emphasise that the benchmarks are not meant to be definitive. We only present them to demonstrate that the performance of our implementation is comparable to state-of-the-art. The results of our tests are presented in table 1. We only report execution times for SMC, but PIMH is almost identical. Comparing speed of PC implementations is more difficult, since our version lets us control the final number of particles, while Anglican and Probabilistic C give control over the starting number of particles. We note that our implementation seems not to scale as well as Anglican with the number of particles. The probable cause for it is that we use an inefficient resampling algorithm, which could be fixed in a later version.

<table>
<thead>
<tr>
<th>model</th>
<th>particles</th>
<th>Haskell</th>
<th>Anglican</th>
<th>Prob-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>HMM</td>
<td>100</td>
<td>0.05s</td>
<td>0.2s</td>
<td>0.1s</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.4s</td>
<td>1.0s</td>
<td>2.9s</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>10s</td>
<td>9s</td>
<td>-</td>
</tr>
<tr>
<td>DP</td>
<td>100</td>
<td>0.05s</td>
<td>0.1s</td>
<td>0.1s</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.4s</td>
<td>0.5s</td>
<td>3.2s</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>6.3s</td>
<td>3.9s</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1. Comparison of efficiency of Sequential Monte Carlo-based inference engines in three different probabilistic programming systems: the one described in this paper (Haskell), Anglican, and Probabilistic C. We report the wall clock time taken to complete a single iteration of SMC with the given number of particles on a single core. We note that the results were largely consistent across different runs, but we do not attempt to rigorously analyse their variability. Those figures are not intended as a definitive evaluation, but rather as a rough indication of relative efficiencies. For Probabilistic C we were unable to run 10000 particles, which requires spawning 10000 processes simultaneously.

5.1 Dice rolling

As a first test we choose the very simple model of rolling dice from section 2.2. We turn it into an inference problem by conditioning it with likelihood inversely proportional to the result.

```haskell
conditionalDie :: Int -> Dist Int
conditionalDie n = condition ((1 /) . fromIntegral) (die n)
```

We use 5 dice and obtain the exact posterior by exhaustively enumerating all possible outcomes. The results clearly show that all the inference methods converge to the exact posterior.

5.2 Hidden Markov Model

The next model is more challenging and practically relevant. We use the HMM, exactly in the form as defined in 2.5. We compute the

\(^1\)https://bitbucket.org/dtolpin/anglican
exact marginal distribution for each latent state using the forward-
backward algorithm. The KL divergence reported is actually a sum
of KL divergences between exact and empirical marginal posteriors
for each latent state. We only include KLS for the latent variables
that have corresponding observations, that is we exclude the initial
state.

5.3 DP mixture of Gaussians

Our final test is the DP mixture model from 2.6. We report the KL
divergence between total posteriors over all the clusters for the
data points presented in 2.6. The exact posterior was computed
by exploiting the conjugate Normal-Inverse-Gamma prior, the ex-
act values of the Chinese Restaurant Process prior, and an enumera-
tion of all possible partitions.

6. Formal semantics

In this section we formally define the semantics of Dist. Two
major technical difficulties involved are defining and conditioning
probability distributions over uncountable sets. We turn to measure
theory for a solution to those problems. Measure theory is a vast
subject and rather than trying to review it here, we refer the readers
to general texts such as (Rosenthal 2006).

We only define semantics for distributions over sufficiently sim-
ple types and under certain simplifying assumptions. Specifically,
we require that all the functions involved always terminate, so we
can treat Haskell functions as total. We also require that there is
a finite number of Conditional points in the GADT and that all
the scoring functions are bounded. Relaxing those requirements
is an important goal, but it is outside the scope of this paper.

Let us consider Dist to be a function from a suitable source of
randomness to a pair consisting of a value and a weight. For this
purpose we define Haskell functions semantics, value, and weight

\[
\text{semantics} \ (\text{Dist} \ a \to \ P \ a) \to \ (\text{Dist} \ a \to \ [\text{Double}] \to \ \text{Maybe} \ (a, \text{Prob}, [\text{Double}])
\]

\[
\begin{align*}
\text{semantics} \ (\text{Bind} \ d) \ tape &= \text{Just} \ (x,1,\text{tape}) \\
\text{semantics} \ (\text{Return} \ x) \ t &= \text{Just} \ (x,1,t) \\
\text{semantics} \ (\text{Primitive} \ []) &= \text{Nothing} \\
\text{semantics} \ (\text{Conditional} \ c \ d) &= \text{do} \\
\text{weight} \ (\text{Bind} \ d) &= \text{fromMaybe} \ 0 \ \text{fmap} \ (\lambda (x,y,z) \to y) \ (\text{semantics} \ d \ t) \\
\text{value} \ (\text{Bind} \ d) &= \text{fromMaybe} \ 0 \ \text{fmap} \ (\lambda (x,y,z) \to x) \ (\text{semantics} \ d \ t) \\
\text{member} \ (\text{Bind} \ d) &= \text{do} \\
\text{member} \ (\text{Primitive} \ []) &= \text{Bool} \\
\text{member} \ (\text{Conditional} \ c \ d) &= \text{do} \\
\text{member} \ (\text{Return} \ x) &= \text{Bool} \\
\end{align*}
\]

We choose a source of randomness to be a finite list of real num-
bers from the [0, 1] interval. This means the source may sometimes
be insufficient, which is why the result is wrapped in a Maybe. On
the other hand, semantics always terminates on Dists that con-
sume randomness indefinitely. The function member tests if the re-
sult belongs to a set defined by an indicator function.

**Theorem 1.** Dist is a monad up to equivalence defined by
\text{semantics}.

\[
\begin{align*}
\text{semantics} \ (\text{Return} \ x) &= \text{Just} \ (x,1,t) \\
\text{semantics} \ (\text{Bind} \ (\text{Return} \ x) f) &= \text{do} \\
& \text{Just} \ (x,1,t) \\
\text{semantics} \ (\text{Bind} \ f \ g) &= \text{do} \\
& (x,p,t) \to \text{semantics} \ f \ x \\
& \text{Just} \ (x,1,t) \\
\text{semantics} \ (\text{Bind} \ f \ \text{Return}) &= \text{do} \\
& (x,p,t) \to \text{semantics} \ f \ x \\
& \text{Just} \ (x,1,t) \\
\end{align*}
\]

Proof. In order to be a monad, Dist needs to satisfy the three
monad laws. Since semantics is defined in a compositional way
by recursion on the structure of Dist, we only need to consider
direct application of semantics to Dists that should be equivalent
by the monad laws.

- **left identity:** return a >>= f = f a

\[
\begin{align*}
\text{semantics} \ (\text{Bind} \ (\text{Return} \ a) f) &= \text{do} \\
& (x,p,t) \to \text{semantics} \ (\text{Return} \ a) \\
& (y,p\cdot q,t') \to \text{semantics} \ (f \ x) \\
& \text{Just} \ (x,1,t) \\
\end{align*}
\]

- **right identity:** m >>= return = m

\[
\begin{align*}
\text{semantics} \ (\text{Bind} \ m \ \text{Return}) &= \text{do} \\
& (x,p,t) \to \text{semantics} \ m \\
& (y,p\cdot q,t') \to \text{semantics} \ (\text{Return} \ a) \\
& \text{Just} \ (x,1,t) \\
\end{align*}
\]

- **associativity:** m >>= f >>= g = m >>= \( \lambda x \to f x \) >>= g

\[
\begin{align*}
\text{semantics} \ (\text{Bind} \ (\text{Bind} \ m \ f) g) &= \text{do} \\
& (x,p,t) \to \text{semantics} \ (\text{Bind} \ m \ f) \\
& (y,p\cdot q,t') \to \text{semantics} \ (g \ x) \\
& \text{Just} \ (x,1,t) \\
\end{align*}
\]
In order to simplify the task of deriving measures from Dist, we restrict the type T in Dist T not to contain any function types. This restriction only applies to the type of the final expression we wish to give semantics to, and not to any intermediate components used for its construction. We do it solely to avoid technical difficulties associated with defining measures on function spaces and products. A type T is simple if it is simple if 0 ∈ T, and is closed under binary sums and products. A type T is simple if 0 ∈ T, and is closed under binary sums and products.

Definition 1. We define the set of simple types S to be the smallest set containing int and real and closed under binary sums and products. A type T is simple if T ∈ S, or equivalently if it is generated by the following grammar:

\[ T ::= \text{int} \mid \text{real} \mid T + T \mid T \times T \]

Our examples in section 2 either define distributions over simple types, or lists of statically known length, which could be easily encoded as a simple type.

Definition 2. For any simple type T we define \([T]\) to be the set of all possible values of this type. This is defined recursively as follows:

\[ [\text{int}] = \mathbb{Z} \]
\[ [\text{real}] = \mathbb{R} \]
\[ [T_1 + T_2] = [T_1] \cup [T_2] \]
\[ [T_1 \times T_2] = [T_1] \times [T_2] \]

where \(\cup\) is a disjoint union and \(\times\) is a Cartesian product.

Definition 3. For any simple type T we define a \(\sigma\)-algebra \(\Sigma_T\) recursively as follows:

\[ \Sigma_{\text{int}} = \sigma([\mathbb{Z}]) \]
\[ \Sigma_{\text{real}} = \sigma([\mathbb{R}]) \]
\[ \Sigma_{T_1 + T_2} = \sigma([T_1 + T_2]) \]
\[ \Sigma_{T_1 \times T_2} = \sigma([T_1 \times T_2]) \]

where \(\sigma(X)\) is the smallest \(\sigma\)-algebra over the set X that is a superset of S.

Together a pair \(([T], \Sigma_T)\) forms a measurable space and we interpret values of type Dist T to be measures over this space.

With the above definitions in place, we proceed to define the semantics in terms of a uniform probability measure on the space of sources of randomness. Since we cannot construct a uniform measure on \([0, 1]^\infty\), we resort to a limit construction involving Borel measures on \([0, 1]^n\) for each finite n.

Definition 4. For any simple type T and any expression M :: Dist T, we define its semantics \([M]\) to be a measure \(\mu\) on \(([T], \Sigma_T)\). For any A ∈ \(\Sigma_T\), \(\mu(A)\) is defined using a limit of Lebesgue integrals. Specifically, for any n ∈ \(\mathbb{N}\), let \(\nu_n\) be a Borel measure on the set \([0, 1]^n\) representing random sources of fixed length n. Then

\[ \mu(A) = \lim_{n \to \infty} \mu_n(A) \]

Proof. Let \(b_n = \lim_{k \to \infty} \alpha_{nk}\) and \(c_k = \lim_{k \to \infty} \alpha_{nk}\). Both \(b_k\) and \(c_k\) are bounded and non-decreasing, so they converge. Let \(b = \lim_{k \to \infty} b_n\), so \(b = \lim_{k \to \infty} \lim_{n \to \infty} \alpha_{nk}\), and let \(c = \lim_{k \to \infty} c_k\), so \(c = \lim_{k \to \infty} \lim_{n \to \infty} \alpha_{nk}\). We have \(\alpha_{nk} \leq c_k\) and taking limits as \(k \to \infty\) from both sides we get \(b_n \leq c\) and so \(b \leq c\). Similarly from \(\alpha_{nk} \leq b_n\) we obtain \(c_k \leq b\) and so \(c \leq b\). It follows that \(\lim_{n \to \infty} \lim_{k \to \infty} \alpha_{nk} = b = c = \lim_{k \to \infty} \lim_{n \to \infty} \alpha_{nk}\).
The source code will be released upon publication.

2The source code will be released upon publication.
convenient and sometimes makes it possible to compose inference algorithms by simply composing the functions implementing them. In particular we obtained Particle Independent Metropolis Hastings by composing Sequential Monte Carlo with Metropolis Hastings. In order to have a notion of correctness of inference algorithms defined this way we defined formal semantics for Dist using measure theory.

We identify three directions for future work. The most straightforward one is to parallelise the implementation. Because we implement inference as a function from Dist a to Dist b, we could at the same time parallelise inference and sampling from a model. The main obstacle is that currently we are only using monads to build Dist and monads are inherently sequential. From the probabilistic point of view, two random variables can be sampled in parallel at the same time parallelise inference and sampling from a model. The implement inference as a function from Dist a to Dist b, we could at the same time parallelise inference and sampling from a model. The main obstacle is that currently we are only using monads to build Dist and monads are inherently sequential. From the probabilistic point of view, two random variables can be sampled in parallel.

Another direction is extending the formal semantics defined in section 6. It would be desirable to extend the definitions to cover recursive types and function types, as well as expressions that may not terminate. In the long run, having Dist with formally defined semantics and inference algorithms that manipulate Dists deterministically, we might hope to formally prove correctness of inference algorithms.

Finally, we plan to investigate the idea of composing inference algorithms in probabilistic programming. The crucial requirement for making such composition really useful is finding more general-purpose inference algorithms that can be composed.

Acknowledgments
A talk by Sriram Rajamani at the Dagstuhl Seminar on Probabilistic Programming helped inspire the semantics in section 6. Discussions with Johannes Borgström, Ugo Dal Lago, Marcin Szymczak, and Ohad Kammar were helpful. David Tolpin and Brooks Paige helped with setting up Anglican and Probabilistic C, respectively. Jeremy Gibbons pointed out that the construction of the HMM presented in section 2.5 is in fact a fold over the data.

References


