Practical Probabilistic Programming with Monads

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

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

Andrew D. Gordon
Microsoft Research, UK
and University of Edinburgh, UK
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 to construct probabilistic models for machine learning, 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. We demonstrate a clean and elegant implementation that achieves performance comparable with Anglican, a state-of-the-art probabilistic programming system.

Categories and Subject Descriptors G.3 [*Probability and Statistics*]: Statistical software

General Terms Languages

Keywords Haskell, probabilistic programming, Bayesian statistics, monads, Monte Carlo

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 θ 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, $P((A, B) \mid (x_i, y_i))$, 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 composeable 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 of them are restricted to ensure that the model has specific properties that make inference easier. For example BUGS (Gilks et al. 1994; Lunn et al. 2013) and Infer.NET (Minka et al. 2009) can only express distributions that can be compiled to a finite graphical model. The Stan language (Stan Development Team 2014) restricts the model parameters to be continuous and can only handle a finite number of them to ensure that inference can be performed with the Hamiltonian Monte Carlo algorithm (Neal 2011). Those approaches simplify writing inference algorithms, but at the price of reduced expressiveness.

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). Others such as HANSEI (Kiselyov and Shan 2009) embed probabilistic DSLs in existing programming languages such as OCaml, but are limited to discrete distributions.

The most general approach, known as *universal* probabilistic programming, allows the user to specify any model with a com-

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

Haskell'15, September 3-4, 2015, Vancouver, BC, Canada
© 2015 ACM. 978-1-4503-3808-0/15/09...
http://dx.doi.org/10.1145/2804302.2804317

putable prior, including some Bayesian nonparametric models such as the Dirichlet Process. 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 harder, and research on improving inference is flourishing (Goodman and Stuhlmüller 2014).

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 unscalable 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 allows for composing implementations of inference algorithms.
- 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. 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. We 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 reduce the risk of underflow.

```
newtype Prob = Prob {toDouble :: Double}
    deriving (Show, Eq, Ord, Num, Fractional, Real,
     RealFrac, Floating, Random)
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 complex models

```
instance Monad Dist where
  return = Return
  (>>=)  = Bind
```

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 -> Dist a
```

Note that `condition` expects an explicit function for computing the likelihood. In practice such a function is usually constructed by taking a `pdf` of a distribution representing the noise model (see section 3.2). Concrete examples are shown later in this section.

4. a way to sample from a `Dist`

```
sample :: StdGen -> Dist 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 :: Int -> Dist Int
die 0 = return 0
die 1 = uniform [1..6]
die n = liftM2 (+) (die 1) (die (n-1))
```

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

```
result   = sample g (die n)
results = sample g $ sequence $ replicate k $ die n
```

An example run would be the following.

```
> let (n,k) = (3,5)
> let g = mkStdGen 0
> sample g $ die n
14
> sample g $ sequence $ replicate k $ 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 `Dist` is lazy, so it is equivalent to write

```
results = take k $ sample g $ sequence $ repeat $ 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 simple example to demonstrate 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. If the coin is biased, we also want to determine its weight. We start by putting a prior over the weight of the coin. The coin is fair with probability 0.8 and biased with probability 0.2. For a biased coin we put a prior on its weight according to a Beta distribution. This is a standard choice that makes it possible to compute the posterior analytically.

```
weight :: Dist Prob
weight = do
  isFair <- bernoulli 0.8
  if isFair then return 0.5 else beta 5 1
```

Now we define the likelihood model. We represent outcomes of a coin toss with Boolean values and we identify the weight of the coin with the probability of landing `True` in a single toss. For a coin with weight w the likelihood is equal to w for the coin landing `True` and $1 - w$ for the coin landing `False`. We define a `toss` function which conditions the distribution over the possible weights of the coin on an outcome of a single toss. Conditioning on a series of outcomes is just a fold over the outcomes.

```
toss   :: Bool -> Dist Prob -> Dist Prob
toss   b    = condition (\w -> if b then w else 1 - w)
tosses :: [Bool] -> Dist Prob -> Dist Prob
tosses bs d = foldl (flip toss) d bs
```

The above functions are a simple example of a recurring idiom for writing models in this style. In more complicated models in subsequent subsections model construction is interleaved with observing data points, so an equivalent of the `toss` function both observes a data point and builds a part of the model.

The posterior over the weight of the coin is:

```
posteriorWeight = tosses observations 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.

```
type Point = (Double, Double)
type Param = (Double, Double)

linear :: Dist Param
linear = do
  a <- normal 0 1
  b <- normal 0 1
  return (a,b)

point :: Point -> Dist Param -> Dist Param
point (x,y) =
  Conditional (\(a,b) -> pdf (Normal (a*x + b) 1) y)
points :: [Point] -> Dist Param -> Dist Param
points ps d = foldl (flip point) d ps
```

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. There are three possible values for each latent state, namely -1, 0, and 1. The observed value is the latent value distorted by Gaussian noise. We observe values of 16 subsequent states, some close to the values of latent states, some far away.

```
hmm :: Dist [Int]
hmm = liftM reverse states where
  states = foldl expand start values
  expand :: Dist [Int] -> Double -> Dist [Int]
  expand d y = condition (score y . head) $ do
    rest <- d
    x    <- trans $ head rest
    return (x:rest)
  score y x = pdf (Normal x 1) y
  trans 0 = categorical $ zip [-1..1] [0.1, 0.4, 0.5]
  trans 1 = categorical $ zip [-1..1] [0.2, 0.6, 0.2]
  trans 2 = categorical $ zip [-1..1] [0.15,0.7,0.15]
  start   = uniform [[-1], [0], [1]]
  values = [0.9,0.8,0.7,0,-0.025,5,2,0.1,0,
            0.13,0.45,6,0.2,0.3,-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) mixture, where the number of clusters is not determined by a specific parameter, but rather can grow with the number of data points. One way to think about it is that rather than selecting the number of clusters and then distributing the points between them, we lazily instantiate an infinite number of clusters and then distribute the points between them.

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.

```
stick :: [Prob] -> [a] -> Dist a
stick (b:breaks) (a:atoms) = do
  keep <- bernoulli b
  if keep then return a else stick 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
      return (classgen, vars, means)
    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 is Gaussian
    score y (cluster, var, mean) =
      -- Normal mean stddev
      prob $ pdf (Normal mean (sqrt var)) y

  in
   --exctract 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 = fst $ randomR (0.0,1.0) g
    scan v ((x,p):ps) =
      if v <= p then x else scan (v-p) ps

uniform     = Explicit . normalize . map (,1)
categorical = Explicit . normalize
bernoulli p = categorical [(True,p), (False,1-p)]
```

3.2 `Sampleable` Type Class

In this work we are mainly concerned with sampling-based inference algorithms. In a pure functional language such as Haskell those algorithms can be implemented as pure functions that take an additional argument which serves as a source of randomness. To specify which data types can be sampled from we introduce the `Sampleable` type class.

```
class Sampleable d where
  sample  :: StdGen -> d a -> a
  sampleR :: Double -> d a -> a
```

The function `sample` uses a standard pseudo-random number generator and is the one that forms the basis of our implementation. The function `sampleR` uses a number from $[0, 1]$ interval as a source of randomness. We do not use it in practice, but it is useful for defining semantics in section 6.

Finally, we import implementation of standard probability distributions such as normal or gamma from the random-fu[1] package. The uppercase constructors such as `Normal` or `Gamma` in this paper are imported from random-fu. Their lowercase counterparts are wrappers that make them `Sampleable`. Apart from sampling, those standard distributions also support computing the probability density function with the following signature:

```
pdf :: d a -> a -> Double
```

3.3 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. One way to do that would be to make `Dist` a type class with different inference algorithms corresponding to different instances of it. However, this implementation would make it difficult to compose inference algorithms in a way that is used in section 4.5. For this reason we implement `Dist` as a GADT which can be identified as a free monad.

```
data Dist a where
  Return      :: a -> Dist a
  Bind        :: Dist b -> (b -> Dist a) -> Dist a
  Primitive   :: Sampleable d => d a -> Dist a
  Conditional :: (a -> Prob) -> Dist a -> Dist a
condition = Conditional

instance Functor Dist where
  fmap = liftM

instance Monad Dist where
  return = Return
  (>>=)  = Bind

instance Sampleable Dist where
  sample g (Return x)        = x
  sample g (Bind d f)        = sample g1 y  where
    y       = f (sample g2 d)
    (g1,g2) = split g
  sample g (Primitive d)     = sample g d
  sample g (Conditional c d) = undefined
```

[1] `https://hackage.haskell.org/package/random-fu`

The `Primitive` constructor can be used with any `Sampleable` type. This includes the basic set of elementary distributions as well as any user-defined distributions.

A `Dist` can only be sampled from if it is not `condition`ed. 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 (approximately) 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 a` into `Dist [a]` or `Dist [(a,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 `Dist a -> 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 emphasise this means the left of `>>=` and not the left of `<-` in a `do` block. In a `do` block this constraint means that all `Conditionals` must be placed on the right of the first `<-`.

The above condition is enforced by the `sample` function which is undefined for the `Conditional` case. The inference algorithms we present only remove `Conditionals` from the left of `Bind`, so if there are any `Conditionals` on the right of it, they will remain after inference and fail when `sample` is called on the result. It would be desirable to enforce this requirement statically through the type system, but unfortunately it would prevent `Dist` from being an instance of the `Monad` class. Specifically, consider the following design.

```
data PDist a where
  Return     :: a -> PDist a
  PBind      :: PDist b -> (b -> PDist a) -> PDist a
  Primitive  :: Sampleable d => d a -> PDist a

data CDist a where
  PD           :: PDist a -> CDist a
  CBind        :: CDist b -> (b -> PDist a) -> CDist a
  Conditional  :: (a -> Prob) -> CDist a -> CDist a
```

In the above `PDist` is a distribution without conditioning, while `CDist` is one with conditioning. `PDist` is a lstinlineMonad, but

unfortunately `CDist` is not, as the signature of `CDist` does not match the signature of `>>=`.

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`.

```
prior' :: Dist a -> Dist a
prior' (Conditional c d) = prior' d
prior' (Bind d f)        = Bind (prior' d) f
prior' d                 = d
```

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.

```
prior :: Dist a -> Dist (a,Prob)
prior (Conditional c d) = do
  (x,s) <- prior d
  return (x, s * c x)
prior (Bind d f) = do
  (x,s) <- prior d
  y     <- f x
  return (y,s)
prior d = do
  x     <- d
  return (x,1)
```

4.2 Importance Sampling

The subsequent methods will often use collections of weighted samples, so we define some helper functions to deal with them.

```
type Samples a = [(a,Prob)]
resample :: Samples a -> Dist (Samples a)
resample xs = sequence $ replicate n $ fmap (,1) $
     categorical xs where
    n = length xs
flatten :: Samples (Samples a) -> Samples a
flatten xss = [(x,p*q) | (xs,p) <- xss, (x,q) <- xs]
```

A simple but useful inference method is that of importance sampling. In essence importance sampling amounts to drawing samples from a tractable proposal distribution (here the prior) and reweighting them in a suitable way (here by the likelihoods). As discussed above, we can either retain all the samples to approximate the posterior or just draw one of them at the end.

```
importance' :: Int -> Dist a -> Dist a
importance' n d = importance n d >>= categorical

normalize :: Samples a -> Samples a
normalize xs = map (second (/ norm)) xs where
```

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. The single-site MH is discussed in section 8.

```
mh :: Dist a -> Dist [a]
mh d =
    fmap (map fst) $ proposal >>= iterate where
        proposal = prior d
        iterate (x,s) = do
          (y,r) <- proposal
          accept <- bernoulli $ min 1 (r / s)
          let next = if accept then (y,r) else (x,s)
          rest  <- iterate next
          return $ next:rest

mh' :: Int -> Dist a -> Dist a
mh' n d = fmap (!! n) (mh d)
```

4.4 Sequential Monte Carlo

There exists a more powerful inference method, somewhat similar to importance sampling, called Sequential Monte Carlo (SMC) (Doucet and Johansen 2011). 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 of `Conditionals` is not affected by any random choices, that is, all `Conditionals` are to the left of `Bind`.

```
smc  :: Int -> Dist a -> Dist (Samples a)
smc n (Conditional c d) = updated >>= resample where
  updated = fmap normalize $
            condition (sum . map snd) $ do
              ps    <- smc n d
              let qs = map (\(x,w) -> (x, c x * w)) ps
              return qs
smc n (Bind d f) = do
  ps <- smc n d
  let (xs,ws) = unzip ps
  ys <- mapM f xs
  return (zip ys ws)
smc n d = sequence $ replicate n $ fmap (,1) d

smc' :: Int -> Dist a -> Dist a
smc' n d = smc n d >>= categorical
```

We emphasize the fact that SMC does not actually remove `Conditionals`, but rather replaces them with different ones. The new scores are weighted averages of likelihood scores across all the particles, which is sometimes called a pseudo-marginal likelihood. It is useful to retain the pseudo-marginal likelihood, since it can be used to correct for bias introduced by SMC. The reason we do it

this way is that it allows us to compose two inference methods to obtain better results. In particular, it is possible to discard the extra information and recover the ordinary SMC algorithm by composing `smc` with `prior'`.

```
smcStandard :: Int -> Dist a -> Dist (Samples a)
smcStandard n = prior' . smc n

smcStandard' :: Int -> Dist a -> Dist a
smcStandard' n = prior' . smc' n
```

Another possibility is to combine results of several SMC runs by weighting the particles according to the pseudo-marginal likelihoods. We can accomplish it by composing `smc` with `importance`.

```
smcMultiple :: Int -> Int -> Dist a -> Dist (Samples a)
smcMultiple k n = fmap flatten . importance k . smc n

smcMultiple' :: Int -> Int -> Dist a -> Dist a
smcMultiple' k n = importance' k . smc' n
```

4.5 Particle Independent Metropolis Hastings

The idea that SMC can be used as a part of a more powerful inference algorithm is not new. In particular, there exist a family of MCMC algorithms, 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`.

```
pimh :: Int -> Dist a -> Dist [Samples a]
pimh n = mh . smc n

pimh' :: Int -> Int -> Dist a -> Dist a
pimh' k n = mh' k . smc' n
```

4.6 Particle Cascade

A final inference algorithm that we discuss here is the recently proposed 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.

```
cascade  :: Dist a -> Dist (Samples a)
cascade (Conditional c d) = do
  ps    <- cascade d
  let qs = map (\(x,w) -> (x, c x * w)) ps
  resamplePC qs
cascade (Bind d f) = do
  ps <- cascade d
  let (xs,ws) = unzip ps
  ys <- mapM f xs
  return (zip ys ws)
cascade d = sequence $ repeat $ fmap (,1) d

cascade' :: Int -> Dist a -> Dist a
cascade' n d = cascade d >>= categorical . take n
```

We do not show the code for the `resamplePC` function here, as it implements a complicated mathematical formula that would be difficult to read. We refer the interested readers to Paige et al. (2014).

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 implementation, not to investigate relative performance of different inference methods. In order to establish correctness we compare the empirical posterior consisting of samples obtained from inference algorithms to an exact posterior. We carefully select models for which we can compute the exact posterior by a combination of analytical 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 [2] 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 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.

model	particles	Haskell	Anglican	Prob-C
HMM	100	0.05s	0.2s	0.1s
	1000	0.4s	1.0s	2.9s
	10000	10s	9s	-
DP	100	0.05s	0.1s	0.1s
	1000	0.4s	0.5s	3.2s
	10000	6.3s	3.9s	-

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.

[2] https://bitbucket.org/dtolpin/anglican

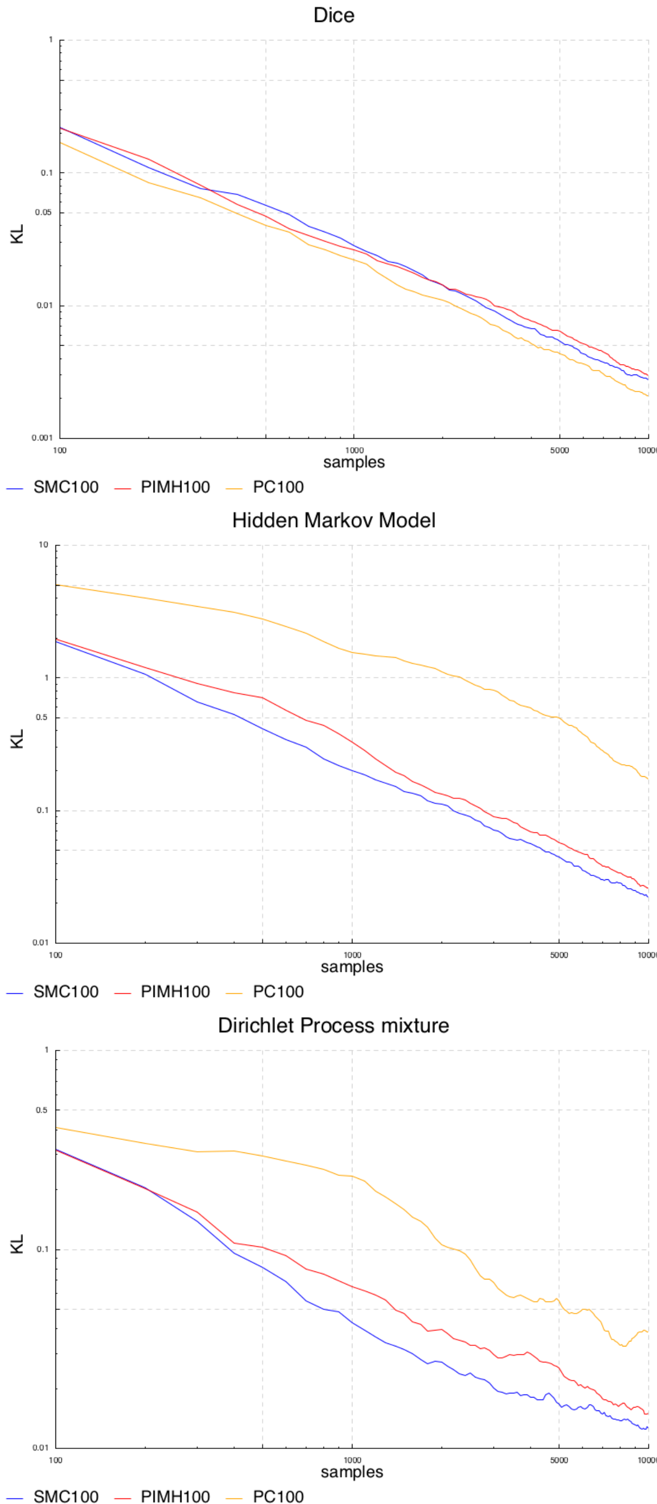

Figure 1. KL divergence between the empirical posterior from samples and the exact posterior as a function of the number of samples drawn for the HMM model. The lines drawn are mean values of KL taken across 20 independent runs. We compare three algorithms: SMC, PIMH, PC, each using 100 particles. SMC and PIMH converge to the correct posterior relatively fast. PC is less efficient per sample drawn, but still displays correct behaviour. The models are described in detail in section 2.

```
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 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 posteriors over the total number of clusters for the data points presented in 2.6. The exact posterior was computed by exploiting the conjugate Normal-Inverse-Gamma prior, the exact values of the Chinese Restaurant Process prior, and an enumeration 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 large 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 simple 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 only a finite number of `Conditional` points in the GADT. 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` and `density`.

```
type P a = [Double] -> Maybe (a, Prob, [Double])

semantics :: Dist a -> P a
semantics (Return x) tape = Just (x,1,tape)
semantics (Bind d f) tape = do
   (x,p,t ) <- semantics d tape
   (y,q,t’) <- semantics (f x) t
   return (y,p*q,t’)
semantics (Primitive d) []     = Nothing
semantics (Primitive d) (r:rs) = Just (sampleR r d,1,rs)
semantics (Conditional c d) tape = do
    (x,p,t) <- semantics d tape
    return (x, p * c x, t)

density :: Dist a -> (a -> Bool) -> [Double] -> Prob
density m i t = case semantics m t of
  Just (x,p,_) | i x -> p
  _                  -> 0
```

We choose a source of randomness to be a finite list of real numbers from the $[0, 1]$ interval. This means the source may sometimes be insufficient, which is why the result is wrapped in a `Maybe`

type. On the other hand, `semantics` always terminates on `Dist`s that consume randomness indefinitely. The function `sampleR` used with `Primitive` is defined in section 3.2. The function `density` approximates the unnormalised posterior density for the specified model using a finite source of randomness.

The definition above can be used to show that `Dist` obeys the monad laws up to equivalence defined by `semantics`. Rather than presenting a tedious but straightforward proof, we show an equivalent, but perhaps more difficult to understand, definition of `semantics'` based on monad transformers. There `P'` is itself a monad, so the proof is trivial.

```
type P' a = ReaderT [Double] (StateT Prob Maybe) a

semantics' :: Dist a -> P' a
semantics' (Return x) = return x
semantics' (Bind d f) = semantics' d >>= semantics' . f
semantics' (Primitive d) = do
  rs <- ask
  case rs of
    []    -> lift $ lift Nothing
    (r:_) -> local tail $ return $ sampleR r d
semantics' (Conditional c d) = do
  x <- semantics' d
  lift $ modify (* c x)
  return x

runDist' :: Dist a -> [Double] -> Maybe (a,Prob)
runDist' d rs = (`runStateT` 1) $ (`runReaderT` rs) $
    semantics' d

density' :: Dist a -> (a -> Bool) -> [Double] -> Prob
density' m i rs = case runDist' m rs of
  Just (x,p) | i x -> p
  _                -> 0
```

To simplify the task of deriving measures from `Dist`s 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 in principle this restriction could be lifted, at least to some degree. Formally, we require T to be either an integer, a real, or constructed by any finite sums and products of the two.

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 $T \in S$, or equivalently if it is generated by the following grammar:*

$$T ::= int|real|T + T|T \times T$$

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:*

$$\begin{aligned}
[\![int]\!] &= \mathbb{Z} \\
[\![real]\!] &= \mathbb{R} \\
[\![T_1 + T_2]\!] &= [\![T_1]\!] \sqcup [\![T_2]\!] \\
[\![T_1 \times T_2]\!] &= [\![T_1]\!] \times [\![T_2]\!]
\end{aligned}$$

where $\sqcup$ is a disjoint union and $\times$ is a Cartesian product.

Definition 3. *For any simple type T we define a σ-algebra Σ_T recursively as follows:*

$$\begin{aligned}
\Sigma_{int} &= \sigma_{\mathbb{Z}}(\mathbb{Z}) \\
\Sigma_{real} &= \sigma_{\mathbb{R}}(\{(a,b)|a,b \in \mathbb{R}\}) \\
\Sigma_{T_1+T_2} &= \sigma_{[\![T_1+T_2]\!]}(\{A \sqcup B | A \in \Sigma_{T_1}, B \in \Sigma_{T_2}\}) \\
\Sigma_{T_1 \times T_2} &= \sigma_{[\![T_1 \times T_2]\!]}(\{A \times B | A \in \Sigma_{T_1}, B \in \Sigma_{T_2}\})
\end{aligned}$$

where $\sigma_X(S)$ is the smallest σ-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 μ on $([\![T]\!], \Sigma_T)$. For any $A \in \Sigma_T$, $\mu(A)$ is defined using a limit of Lebesgue integrals. Specifically, for every $n \in \mathbb{N}$, let ν_n be a Borel measure on the set $[0,1]^n$ representing random sources of fixed length n. Then*

$$\begin{aligned}
\mu(A) &= \frac{\lim_{n\to\infty} \mu_n(A)}{\lim_{n\to\infty} \mu_n([\![T]\!])} \\
\mu_n(A) &= \int (density\ M\ A)\, d\nu_n
\end{aligned}$$

In the above definition, in $density\ M\ A$ we are writing A for its indicator function. This definition is only valid if the denominator is finite, otherwise we consider the model to be misspecified. The denominator can not be 0, so any zero-measure observations need to be noisy. See section 2.1 for more details about conditioning.

An important technicality to note is that definition 4 implicitly assumes the function $density\ M\ A$ to be measurable. We do not prove this, but we claim that this restriction is not a problem in practice and we conjecture that it is in fact not possible to define in Haskell a non-measurable between Euclidean spaces. The question of measurability only becomes an issue when we consider giving semantics to more complex terms, which we do not attempt here.

We leave for future work allowing for more sophisticated types to be used at top level. This includes recursive types, such as lists with potentially infinite, lazily evaluated values, and function types, including higher-order functions.

7. Related Work

The work on semantics of probabilistic programming languages dates back to Dexter Kozen (1981, 1985).

We believe that our formal semantics is the first detailed semantics for a higher-order probabilistic programming language with conditioning, an essential ingredient of Bayesian modelling. Ramsey and Pfeffer (2002) propose formal semantics of a stochastic λ-calculus in terms of probability monads and measure terms, but without considering conditioning. Borgström et al. (2013) develop a compositional semantics of first-order functional programs, including conditioning on zero-probability events, based on measure transformers, but they do not consider the higher-order case. In recent work, van de Meent et al. (2015) provide a formal operational semantics for a Church-like language to associate expressions with execution traces, so as to explain an inference method, Particle Gibbs with Ancestor Sampling, as opposed to the semantics of the Bayesian model. Instead, we have expressed various inference methods as executable Haskell code, and our formal semantics captures the meaning of a Bayesian model directly as a probability dis-

tribution. Some recent works develop operational equivalence for probabilistic λ-calculus but without considering continuous distributions, or constructs for conditioning. Dal Lago et al. (2014) develop a bisimulation theory for an untyped calculus, while Bizjak and Birkedal (2015) investigate operational equivalence and step-indexed logical relations for a typed probabilistic λ-calculus. Gretz et al. (2015) consider the semantics of conditioning in probabilistic while-programs, together with transformations to eliminate conditioning, but without higher-order functions.

A monadic Haskell library for probabilistic programming programming was first introduced by Erwig and Kollmansberger (2006). Later Larsen (2011) suggested using GADTs to improve performance of such a library. Both of these approaches, however, were limited to problems that can be solved by exhaustively enumerating all possible outcomes.

Many different probabilistic programming systems were proposed as practical tools for Bayesian modelling and inference and it is not possible to list them all here. Perhaps the first widely influential one was BUGS (Gilks et al. 1994), which performs Gibbs sampling on finite graphical models. Restricting models to those that can be represented by finite graphs allows for implementation of efficient inference algorithms and was successfully leveraged by systems such as Infer.NET (Minka et al. 2009) and Stan (Stan Development Team 2014). Those tools are fast and practical, but they do not target as wide a range of models as our approach.

A lot of work has been put into combining graphical models with first-order logic (Getoor and Taskar 2007), which extends the set of representable models. Some of the languages in this category include BLOG (Milch et al. 2005), Markov Logic Networks (Domingos and Richardson 2004), ProbLog (Kimmig et al. 2011), and IBAL (Pfeffer 2001). Those languages are more expressive than graphical models, but they still can not express models such as the Dirichlet Process. An approach similar to ours was taken by Kiselyov and Shan (2009) in HANSEI, which embeds a probabilistic programming language in OCaml. However, even though it can compose distributions in arbitrary ways, it is still limited to discrete distributions.

The idea of a universal probabilistic programming language, which can handle both discrete and continuous distributions and compose them in arbitrary ways, was pioneered by Goodman et al. (2008) in Church. Even though very expressive, Church struggles with making inference efficient. Several Church-inspired languages were invented in an attempt to make inference faster, such as Venture (Mansinghka et al. 2014), Anglican (Wood et al. 2014), and Probabilistic C (Paige and Wood 2014). Anglican was first to implement, in the context of probabilistic programming, the SMC-based inference methods used in this paper. Our work differs from Anglican by embedding a probabilistic programming language in Haskell and implementing inference as a deterministic data structure transformation.

Fun (Gordon et al. 2013; Bhat et al. 2013) is an embedding of probabilistic programming in F#. Probabilistic models are quoted F# expressions, instead of being of monadic type. Inference is achieved by parsing the quoted expressions and translating to engines such as Infer.NET.

Pfeffer (2009) describes a probabilistic object-oriented programming language called Figaro, which builds probabilistic programs as data structures in Scala. It targets Metropolis-Hastings with custom proposals, rather than SMC, as the primary inference methods. Another probabilistic programming language embedded in Scala is WOLFE (Riedel et al. 2014). WOLFE goes beyond stating models via generative processes (as in our work), and allows the programmer to compose a model in terms of scalar objectives or scoring functions, and operations such as maximization and summation. WOLFE can express conditional random fields, Markov logic networks, and matrix factorization, but has not been applied to sequential Monte Carlo methods.

8. Conclusion and Future Work

The main focus of this work was on making probabilistic programming with monads practical by implementing powerful approximate inference algorithms. We accomplished this goal by defining a GADT `Dist` representing probability distributions and implementing Sequential Monte Carlo and Particle Markov Chain Monte Carlo algorithms to perform inference on it. We showed that our implementation is competitive with Anglican, the original implementation of those algorithms in a non-monadic probabilistic programming setting. All of our code is freely available as a Haskell library [3].

Apart from using monads, a distinctive feature of our framework is that randomised inference algorithms can be implemented as deterministic transformations on a data structure representing a distribution. This approach makes the implementation clean and 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. We also define formal semantics for `Dist` using measure theory. This semantics could be used to define a notion of correctness for inference algorithms.

We identify four directions for future work. The most straightforward one is to parallelise the implementation. Because we implement inference as a function from `Dist` to `Dist`, 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 `Dists` and monads are inherently sequential. From the probabilistic point of view, two random variables can be sampled in parallel if they are independent conditionally on what was sampled earlier. We may enable parallel sampling if we introduce additional `Dist` constructors that make conditional independence explicit, such as `Independent :: Dist a -> Dist b -> Dist (a,b)`.

An interesting problem is how to extend the `Dist` GADT to allow for implementation of other successful general-purpose inference algorithms. The form of `Dist` presented in this paper is very simple and does not expose all the important information about the model. For example we could not implement the single-site Metropolis-Hastings proposed by Wingate et al. (2011), the GADT does not contain the lexical information needed to tag different random choices in the model. This issue could be overcome by introducing another constructor for structure-modifying random choices. An alternative would be to do a separate preprocessing step on the Haskell source code to capture the required information.

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 `Dists` with formally defined semantics and inference algorithms that manipulate `Dist` s deterministically, we might hope to formally prove correctness of inference algorithms. It would also be interesting to develop a rigorous mechanized form of our semantics, building on the work of Eberl et al. (2015).

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.

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

Acknowledgments

We would like to thank Chung-chieh Shan for reviewing the paper, pointing out errors in section 6 and suggesting fixes for them. We would also like to thank the anonymous reviewers for many useful comments that helped us improve the paper.

A talk by Sriram Rajamani at the Dagstuhl Seminar on Probabilistic Programming helped inspire the semantics in section 6. Discussions with Johannes Borgström, Manuel Eberl, Ugo Dal Lago, Johannes Hölzl, Ohad Kammar, Tobias Nipkow, Marcin Szymczak and 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.

The first author is supported by EPSRC and the Cambridge Trust.

References

C. Andrieu, A. Doucet, and R. Holenstein. Particle markov chain monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010. ISSN 1467-9868. . URL `http://dx.doi.org/10.1111/j.1467-9868.2009.00736.x`.

D. Barber. *Bayesian Reasoning and Machine Learning*. CUP, 2012.

S. Bhat, J. Borgström, A. D. Gordon, and C. V. Russo. Deriving probability density functions from probabilistic functional programs. In *TACAS*, pages 508–522, 2013.

A. Bizjak and L. Birkedal. Step-indexed logical relations for probability. In A. M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2015. . URL `http://dx.doi.org/10.1007/978-3-662-46678-0_18`.

J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. V. Gael. Measure transformer semantics for Bayesian machine learning. *Logical Methods in Computer Science*, 9(3), 2013. preliminary version at ESOP'11.

U. Dal Lago, D. Sangiorgi, and M. Alberti. On coinductive equivalences for higher-order probabilistic functional programs. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 297–308. ACM, 2014. . URL `http://doi.acm.org/10.1145/2535838.2535872`.

P. Domingos and M. Richardson. Markov logic: A unifying framework for statistical relational learning. In *Proceedings of the ICML-2004 Workshop on Statistical Relational Learning and its Connections to other Fields*, pages 49–54, 2004.

A. Doucet and A. M. Johansen. A tutorial on particle filtering and smoothing: fifteen years later, 2011.

A. Doucet, S. Godsill, and C. Andrieu. On sequential monte carlo sampling methods for bayesian filtering. *STATISTICS AND COMPUTING*, 10(3): 197–208, 2000.

M. Eberl, J. Hölzl, and T. Nipkow. A verified compiler for probability density functions. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015*, volume 9032 of *Lecture Notes in Computer Science*, pages 80–104. Springer, 2015. . URL `http://dx.doi.org/10.1007/978-3-662-46669-8_4`.

M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16(1):21–34, 2006.

L. Getoor and B. Taskar. *Introduction to Statistical Relational Learning*. The MIT Press, 2007.

W. R. Gilks, A. Thomas, and D. J. Spiegelhalter. A language and program for complex Bayesian modelling. *The Statistician*, 43:169–178, 1994.

N. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI'08)*, pages 220–229. AUAI Press, 2008.

N. D. Goodman. The principles and practice of probabilistic programming. In *Principles of Programming Languages (POPL'13)*, pages 399–402, 2013.

N. D. Goodman and A. Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2015-5-22.

A. D. Gordon, M. Aizatulin, J. Borgström, G. Claret, T. Graepel, A. Nori, S. Rajamani, and C. Russo. A model-learner pattern for Bayesian reasoning. In *POPL*, 2013.

A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Future of Software Engineering (FOSE 2014)*, pages 167–181, 2014.

F. Gretz, N. Jansen, B. L. Kaminski, J. Katoen, A. McIver, and F. Olmedo. Conditioning in probabilistic programming. *CoRR*, abs/1504.00198, 2015. URL `http://arxiv.org/abs/1504.00198`.

C.-K. Hur, A. Nori, S. Rajamani, S. Samuel, and D. Vijaykeerthy. Implementing a correct sampler for imperative probabilistic programs. 2015. Unpublished draft.

A. Kimmig, B. Demoen, L. D. Raedt, V. S. Costa, and R. Rocha. On the implementation of the probabilistic logic programming language problog. *TPLP*, 11(2-3):235–262, 2011. . URL `http://dx.doi.org/10.1017/S1471068410000566`.

O. Kiselyov and C. Shan. Embedded probabilistic programming. In *Conference on Domain-Specific Languages*, pages 360–384, 2009.

D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

D. Kozen. A probabilistic PDL. *J. Comput. Syst. Sci.*, 30(2):162–178, 1985.

K. F. Larsen. Memory efficient implementation of probability monads. unpublished, 2011.

D. Lunn, C. Jackson, N. Best, A. Thomas, and D. Spiegelhalter. *The BUGS Book*. CRC Press, 2013.

V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *Probabilistic, Logical and Relational Learning — A Further Synthesis*, 2005.

T. Minka, J. Winn, J. Guiver, and A. Kannan. Infer.NET 2.3, Nov. 2009. Software available from `http://research.microsoft.com/infernet`.

R. M. Neal. Probabilistic inference using Markov chain Monte Carlo methods. Technical Report CRG-TR-93-1, Dept. of Computer Science, University of Toronto, September 1993.

R. M. Neal. Mcmc using hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 2, 2011.

B. Paige and F. Wood. A compilation target for probabilistic programming languages. In *ICML*, 2014.

B. Paige, F. Wood, A. Doucet, and Y. W. Teh. Asynchronous anytime sequential monte carlo. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3410–3418. Curran Associates, Inc., 2014. URL `http://papers.nips.cc/paper/5450-asynchronous-anytime-sequential-monte-carlo.pdf`.

A. Pfeffer. IBAL: A probabilistic rational programming language. In B. Nebel, editor, *International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 733–740. Morgan Kaufmann, 2001.

A. Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165, 2002.

S. R. Riedel, S. Singh, V. Srikumar, T. Rocktäschel, L. Visengeriyeva, and J. Noessner. WOLFE: strength reduction and approximate programming for probabilistic programming. In *Statistical Relational Artificial Intelligence*, 2014.

J. S. Rosenthal. *A First Look at Rigorous Probability Theory*. World Scientific, 2nd edition, 2006.

Stan Development Team. Stan: A C++ library for probability and sampling, version 2.2, 2014. URL `http://mc-stan.org/`.

Y. W. Teh. Dirichlet processes. In *Encyclopedia of Machine Learning*. Springer, 2010.

J. van de Meent, H. Yang, V. Mansinghka, and F. Wood. Particle gibbs with ancestor sampling for probabilistic programs. In G. Lebanon and S. V. N. Vishwanathan, editors, *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2015, San Diego, California, USA, May 9-12, 2015*, volume 38 of *JMLR Proceedings*. JMLR.org, 2015. URL `http://jmlr.org/proceedings/papers/v38/vandemeent15.html`.

D. Wingate, A. Stuhlmueller, and N. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th Intl. Conf. on Artificial Intelligence and Statistics*, page 131, 2011.

F. Wood, J.-W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, 2014.