Parameterized probability monad

Adam Ścibior
University of Cambridge
and MPI Tübingen

Andrew D. Gordon
Microsoft Research
and University of Edinburgh

Abstract

Probability monads are an attractive tool both for specifying semantics of probabilistic programs and for implementing embedded probabilistic programming languages. We show that parameterized monads can be used to ensure additional properties of distributions constructed by probabilistic programs. As a demonstration we provide a Haskell implementation of a parameterized probability monad that ensures all conditioning is performed statically.

1. Introduction

Probability monads (Giry 1982) have long been used for defining semantics of probabilistic programs (Jones and Plotkin 1989; Ramsey and Pfeffer 2002), and more recently for implementation (Erwig and Kollmansberger 2006; Ścibior et al. 2015) of probabilistic programming languages (PPLs). We explore how the use of a generalization of monads, called parameterized monads, can ensure additional properties of distributions constructed with the monad abstraction.

We start with a definition of a particular implementation of the parameterized monad abstraction we use in this work. Then we show our implementation of the parameterized probability monad and discuss the static guarantees it provides. Finally we define semantics of our implementation based on measure theory, and conclude with suggesting further uses for parameterized monads in probabilistic programming.

2. Parameterized monads

A monad defines two operations, called `return` and `>>=`, which in Haskell corresponds to the following typeclass:

```
class Monad m where
  return :: a -> m a
  >>= :: m a -> (a -> m b) -> m b
```

A parameterized monad is a generalization which permits greater flexibility in the types of arguments that those operations accept. We use the implementation provided by the monad-param library[1], which uses the following typeclasses:

```
class Return m where
  returnM :: a -> m a

class Bind m m' m'' | m m' -> m'' where
  (>>=) :: m a -> (a -> m' b) -> m'' b
```

This definition of `>>=` allows any data types to be used. However, in order to sequence the applications of `>>=`, the result type of the first one must match the argument type of the second one.

[1] `https://hackage.haskell.org/package/monad-param`

3. Parameterized probability monad in Haskell

We improve on (Ścibior et al. 2015) by statically enforcing two requirements which were earlier checked dynamically. The first is that the function `sample` can only be applied to `Dist`s that do not contain conditioning operations. The second is that conditioning is only allowed in the first argument of `>>=` and not in the second. Below we present our new implementation and explain those points in detail.

We use two GADTs to represent probability distributions (or, equivalently, probabilistic programs). The first is for 'pure' distributions without any conditioning statements

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

instance Return Dist where
  returnM = Return

instance Bind Dist Dist Dist where
  (>>=) = Bind
```

and the second one for distributions that may include conditioning

```
data CDist a where
  Pure     :: Dist a -> CDist a
  CBind    :: CDist b -> (b -> Dist a) -> CDist a
  Conditional:: (a -> Prob) -> CDist a -> CDist a

instance Return CDist where
  returnM = Pure . returnM

instance Bind CDist Dist CDist where
  (>>=) = CBind
```

In the spirit of (Ścibior et al. 2015) we consider `Dist` to be a tractable representation and we can easily sample from it. Specifically, we define the following function:

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

However, `CDist` is a declarative description of a distribution which does not specify how to sample from it. To sample from `CDist` we must first convert it into an equivalent `Dist`. Indeed, in this framework we view inference algorithms, including randomized ones, as deterministic functions from `CDist` to `Dist`. For more details we refer the reader to (Ścibior et al. 2015). With the definitions above, we can statically ensure that a suitable inference algorithm is applied before sampling is performed.

The restriction on the `Bind` instance of `CDist` enforces that the observations do not change depending on what random choices were made in the program. In standard machine learning tasks this condition is always satisfied, so it does not limit the expressiveness of the modelling framework. On the other hand, it brings two important benefits:

1. It is possible to sample from the distribution lazily, as it is guaranteed that conditioning is absent from the dynamically generated parts of the model. Otherwise those parts could have global influence on the sampling process, so the samples would have to be generated eagerly.
2. Sequential Monte Carlo can be used for inference with each conditioning point serving as a synchronization barrier (Wood et al. 2014). Without the restriction, the algorithm might get stuck if a particular barrier is not reached by some particles.

The only downside to this restriction is that programming with `CDist` is somewhat less natural. For example, in a simple linear regression example, the following would not be possible[2], since conditioning would now be placed on the right of `>>=`.

```
wrong :: Dist (Double,Double)
wrong = do
  a <- normal 0 1
  b <- normal 0 1
  condition (pdf (normal x 1) y)
  return (a,b)
```

Instead, we need to resort to a less intuitive form, where the prior distribution is explicitly passed as an argument to `condition`.

```
right :: Dist (Double,Double)
right = condition (\(a,b) -> pdf (normal x 1) y)
        d where
  d = do
    a <- normal 0 1
    b <- normal 0 1
    return (a,b)
```

4. Measure-theoretic semantics

We define semantics for `Dist` and `CDist` using measure theory. This definition is very similar to (Ścibior et al. 2015) and the same assumptions are in place. In particular we stress that this semantics only works for the top-level distribution over sufficiently simple types, which excludes function types and recursive types. Distributions over those types can, however, be used at intermediate stages.

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

class ProbMeasure d where
  semantics :: d a -> P a

instance ProbMeasure Dist where
  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)

instance ProbMeasure CDist where
  semantics (Pure d) tape = semantics d tape
  semantics (CBind d f) tape = do
    (x,p,t)  <- semantics d tape
    (y,q,t') <- semantics (f x) t
    return (y,p*q,t')
  semantics (Conditional c d) tape = do
    (x,p,t) <- semantics d tape
    return (x, p * c x, t)

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

Most of the work is done by the `semantics` function, defined both for `Dist` and `CDist`. It converts its input to a function from a source of randomness (a finite list of real numbers) to a value and associated weight, or `Nothing` if the source is too short. This allows us to define the measure by integrating against all sources of randomness. Specifically, for any value `M :: Dist T` or `M :: CDist T` we define a measure μ as

$$\mu(A) = \frac{\lim_{n\to\infty} \mu_n(A)}{\lim_{n\to\infty} \mu_n(\llbracket T \rrbracket)}$$

$$\mu_n(A) = \int (density\ M\ A)\, d\nu_n$$

where $\llbracket T \rrbracket$ is a σ-algebra corresponding to type T (Ścibior et al. 2015).

This definition could potentially be used to establish correctness of inference algorithms, by requiring that the input `CDist` and the result `Dist` correspond to the same measure under the semantics above.

As mentioned above, this definition is only applicable to sufficiently simple types T. It would be desirable to extend it to function and recursive types.

5. Other uses of parameterized monads

In the above we have demonstrated how to utilise parameterized monads to ensure that conditioning is performed statically in probabilistic programs. However, there can be many more potential uses for them. As particularly interesting we point out restricting the latent space of a probabilistic program and keeping track of types of data used for conditioning.

References

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

M. Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 68–85. Springer Berlin Heidelberg, 1982.

C. Jones and G. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 186–195, 1989.

N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2002.

A. Ścibior, Z. Ghahramani, and A. D. Gordon. Practical probabilistic programming with monads. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, pages 165–176. ACM, 2015.

F. Wood, J.-W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In S. Kaski and J. Corander, editors, *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pages 1024–1032. JMLR, 2014.

[2] assuming here fictional `condition :: Prob -> CDist ()`