Probability Monad

Jaime Santos

August 2020

1 Probability Monad

In category theory, the probability monad consists in equipping a category of spaces with a monad whose functor assigns to each space X a space PX of measures, probability measures or valuations [1].

Algebras of probability and measure monads can be intepreted as generalized *convex spaces*, which is a set equipped with a notion of taking weighted averages of its elements [2].

The unit of the monad gives, for each space of outcomes, a map $X \to PX$. In the distribution monad, for example, this type of Kleisli morphisms are stochastic maps. The Kleisli composition of these maps recovers the conditional probability formula [3]

$$p(z|x) = \sum_y p(z|y)p(y|x). \tag{1}$$

The multiplication map of the monad is a map $PPX \to PX$ for all objects X, that can be seen as averaging the probability measures.

1.1 Distribution Monad

This monad assigns the space of finite distributions to each set. It is the finitary prototype of the Probability Monad.

A finite distribution can be defined as the set DX whose elements are functions $p : X \to [0, 1]$ so that $p(x) \neq 0$ for a finite number of x and the sum of all probabilities must equal 1

$$\sum_{x \in X} p(x) = 1. \tag{2}$$

Given a measurable function $f : X \to Y$, one defines the *pushforward* $Df : DX \to DY$, which is how we are able to make our monad functorial. This means that D is an endofunctor on Set [4].

The unit map $\delta : X \to DX$ maps an element that belongs to X to the function $\delta_x : X \to [0, 1]$

$$\delta_x(y) = \begin{cases} 1, & \text{if } y = x \\ 0, & \text{if } y \neq x \end{cases} \tag{3}$$

The multiplication map $E : DDX \to DX$ maps $\eta \in DDX$ to the function $E\eta \in DX$

$$E\eta(x) = \sum_{p \in DX} p(x)\eta(p) \tag{4}$$

Both of these maps satisfy the usual monad laws, and the resulting monad (D, E, δ) is known as the **distribution monad**.

2 Haskell Implementation

Probabilistic events generally imply that the outcome of a measurement is not clear. This seems incompatible with the notion in functional programming that functions return well-defined values, but can be reconciled if probabilistic values are encapsulated in a data type. This paradigm is explored by the authors of [5], who also created the *Haskell* library used in this paper [6].

This new data type is based on representing *distributions*, which means representing the outcome of a probabilistic event

```
newtype T prob a = Cons {decons :: [(a,prob)]}.
```

All functions for building distribution values ensure that the sum of all probabilities of a non-empty distribution is 1. It follows that values of type *T prob* represent the complete set of elements of some probabilistic event.
Building these distributions is done through *spread* functions, that take in list values and return a distribution value

```
type Spread prob a = [a] -> T prob a
```

One such function could be used to represent the outcome of die rolls, which is an uniform distribution of all faces on a die

```
uniform :: Fractional prob => Spread prob a
uniform = shape(const 1)
shape:: Fractional prob => (prob -> prob) -> Spread prob a
```

where *shape* is a function that constructs a distribution based on some function (*const* for uniform, *id* for linear, etc.). So if we were to represent the distribution for a die with 6 faces, we would write

```
d6 = uniform [1..6]
```

In order to extract a probability from a distribution, we define a function **??** that takes an *Event*, which is a function on a distribution

```
type Event a = a->Bool

(??) :: Num prob => Event a -> T prob a -> prob
(??) p = sumP . filter (p . fst) . decons

sumP:: Num prob => [(a,prob)] -> prob
sumP = sum . map snd
```

The next step is to figure out how to combine distributions. There are two ways of doing this, if the events are independent we can just combine all possible occurrences while multiplying their probabilities

```
joinWith :: (a->b->c) -> T prob a -> T prob b -> T prob c
joinWith f (Cons d)(Cons d') = Cons [(f x y, p*q) | (x,p) <- d, (y,q) <- d']
```

If the second event depends on the first, however, it will be a function of type *a -> T prob b* whereas the first event is of type *T prob a*. This type of combination is "simply" the binding operation, if we instantiate our type as a monad

```
instance Num prob => Monad (T prob) where
   return   = certainly
   d >>= f  = Cons [(y,q*p) | (x,p) <- decons d, (y,q) <- decons (f x)]

certainly :: Num prob => a -> T prob a
certainly x = Cons [(x,1)]
```

We can also define the monadic composition of a list of functions as

```
compose :: Monad m => [a -> m a] -> a -> m a
compose = foldl (>=>) return

(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

The definition of this monad allows us to select elements from a collection without putting them back, which causes later selections to be dependent on earlier ones. In most applications, the elements that remain after selection are of no interest. It is then useful to define a Functor instance for the *T prob* type

```
instance Functor (T prob) where
  fmap f (Cons d) = Cons [(f x,p) | (x,p) <- d]
```

which allows us to apply functions to the distributions without altering their structure redundantly. This package also includes an Applicative instance for this type, which allows the application of functions embbeded in the structure

```
instance Num prob => Applicative (T prob) where
  pure     = certainly
  fm <*> m = Cons [(f x,q*p) | (f,p) <- decons fm, (x,q) <- decons m]
```

The last fundamental concept used in this package is the *probabilistic function* which maps values into distributions

```
type Trans prob a = a -> T prob a
```

With all the basic tools defined, we will next analyse a few examples.

2.1 Flipping a Coin

In the spirit of random walks, the first example given here is a toy model of the process of flipping a coin. Firstly, we define the types needed for this

```
data Coin = Heads | Tails deriving (Eq,Ord,Show)
type Probability = Rational
type Dist = T Probability
```

Our coin type can only be heads or tails, and it will be later combined with a probability in the *Dist* type.

```
flipOnce :: Dist(Coin,Coin)
flipOnce = joinWith (,) coin coin
```

Or, equivalently, through the monadic lift

```
flipOnce :: Dist (Coin, Coin)

flipOnce = liftM2 (,) coin coin

Prelude> twoDice
fromFreqs [((Heads,Heads),1 % 4),((Heads,Tails),1 % 4),((Tails,Heads),1 % 4)
,((Tails,Tails),1 % 4)]
```

Flipping twice can be achieved with

```
flipTwice :: Dist (Coin,Coin,Coin)
flipTwice = liftM3 (,,) coin coin coin
```

And a general implementation for events similar to this is given in the next example.

2.2 Boys in a family

In this example we try to calculate the probability of having two boys in a family of two children, knowing that there already is a boy in the family. Firstly we define the types of the children and the family

```
data Child = Boy | Girl
          deriving (Eq,Ord,Show)

type Family = (Child, Child)
```

Given that there is a birth in the family, the probability of it either being a boy or a girl is $\frac{1}{2}$ so we define

```
birth :: Dist Child
birth = uniform [Boy, Girl]
```

This family can then be described as the combination of all possible combinations of two births, since we defined this family of being composed by two children

```
family :: Dist Family
family = liftM2 (,) birth birth

>Prelude family
fromFreqs [((Boy,Boy),1 % 4),((Boy,Girl),1 % 4),((Girl,Boy),1 % 4),((Girl,Girl),1 % 4)]
```

We now need to define a function that generates the conditional probability of a family knowing that that there is at least one boy

```
famWithBoy :: Dist Family
famWithBoy = oneBoy ?=<< family where
    oneBoy(x0,x1) = (x0 == Boy || x1 == Boy)

>Prelude famWithBoy
fromFreqs [((Boy,Boy),1 % 3),((Boy,Girl),1 % 3),((Girl,Boy),1 % 3)]
```

Where ?=« is a monadic combinator for conditional probability, identical to this monad's filter

```
(?=<<) :: (Fractional prob) => (a -> Bool) -> T prob a -> T prob a
(?=<<) = filter
```

The probability of a family of having a boy, knowing that the first child was a boy, is simply

```
famWithTwoBoys :: Probability
famWithTwoBoys = allBoys ?? famWithBoy where
    allBoys(x0,x1) = (x0 = Boy && x1 = Boy)

>Prelude famWithTwoBoys
1 % 3
```

2.3 Rolling the dice

Firstly, we define the basic types we need for this example

```
type Die = Int
type Probability = Rational
type Dist = T Probability
```

An integer to represent the faces of the dice, a rational number for the probability of a given face being seen and distribution type for the outcomes.
A single 6-faced die can be represented, as shown above, by the uniform distribution:

```
die:: Dist Die
die = uniform[1..6]

Prelude> die
fromFreqs [(1,1 % 6),(2,1 % 6),(3,1 % 6),(4,1 % 6),(5,1 % 6),(6,1 % 6)]
```

For a function that can roll n dice, we can use

```
replicateM :: Applicative m => Int -> m a -> m [a]
```

which performs actions n times, gathering the results.

```
dice :: Int -> Dist [Die]
dice = flip replicateM die
```

We're then replicating die n times and combining them into a single structure. The flip is used so partial definition of the function is possible.

Another function we might think of is one that tells us the probability of seeing a certain face, a number of times in a row

```
faceFreq :: Int -> Int -> Int -> Probability
faceFreq p n k = ((==p) . length . filter(==k)) ?? dice n
```

So the probability of getting face 6 twice in 2 dice (or by throwing a dice twice) is

```
Prelude> faceFreq 2 2 6
1 % 36
```

If we're interested in the sum of two faces after rolling a die twice, we can define

```
addTwo :: Dist Die
addTwo = liftM2 (+) die die

>Prelude addTwo
fromFreqs [(7,1 % 6),(6,5 % 36),(8,5 % 36),(5,1 % 9),(9,1 % 9),(...),(12,1 % 36)]
```

References

[1] NLab. monads of probability, measures, and valuations. ncatlab.org/nlab/show/monads+of+probability%2C+measures%2C+and+valuations.

[2] NLab. convex space. ncatlab.org/nlab/show/convex+space.

[3] Michele GIRY. A categorical approach to probability theory. ncatlab.org/nlab/show/pushforward+measure.

[4] NLab. pushforward measure. ncatlab.org/nlab/show/pushforward+measure.

[5] E. Martin and K. Steve. Functional pearls - probabilistic functional programming in haskell. 2006.

[6] E. Martin and K. Steve. Probabilistic functional programming. *Hackage*, 2006. hackage.haskell.org/package/probability-0.2.7.