Unpublished, August 2011

Memory Efficient Implementation of Probability Monads

Functional Pearl

Ken Friis Larsen

Department of Computer Science, University of Copenhagen
kflarsen@diku.dk

Abstract

It is convenient to use monads to make a domain-specific library for working with probabilistic models and computations. However, representing the computed distributions efficiently can be challenging. The straightforward way of representing a discrete distribution as a list of possible outcomes paired with their probability can lead to a humongous representation. We show how to use a symbolic representation of distributions that allows us to compute exact expected values or make simulations, while keeping the memory usage low and without losing the nice monadic interface and algebraic properties.

1. Introduction

Working with probability distributions can be tricky and hard to get right. Fortunately it is possible to make a nice embedded probabilistic language as a monadic library that makes it simple to specify probabilistic models, which can then either be used to compute exact distributions or can be used for approximate simulations.

The core of such a monadic library is the representation of *distributions*, the usual choice is to represent a distribution as a list of outcomes paired with their likelihood see for instance [Filinski 1996, chap. 4] and [Erwig and Kollmansberger 2006]. To recap the Haskell implementation from [Erwig and Kollmansberger 2006]:

```
type Probability = Float
newtype Dist a = D {unD :: [(a,Probability)]}
```

where we represent probabilities as floating point number between 0 and 1; `Dist a` is a distribution over elements of type `a`; and we have the invariant the probabilities in the list should sum up to 1.

With distribution represented as lists it is deceivingly elegant to define a probability monad, that is to make `Dist` an instance of `Monad`, likewise we can make `Dist` an instance of `Functor` and `Applicative`, see Figure 1.

Figure 1 also contains the functions `choice` and `uniform` for building distributions and the function `expected` for computing the expected value of a disribution given a valuation function.

2. The Problem

The first thing that should worry us, as noted in [Filinski 1996], is that `Dist` does not uniquely define distributions. That is, there are no canonical `Dist` representation for a given distribution. Thus, the monadic laws does not hold if we compare representations. However, if we consider `Dist` an abstract type that can only be observed by calling `expected`, or if the elements in the distribution allow ordering we can make a function `toSorted`:

```
toSorted :: Ord a => Dist a → [(a, Probability)]
```

that return a sorted list where equal elements have been collated, thus forming a canonical representation. We shall sometimes call this process *normalisation*. Now the monadic laws holds up to equivalence of what can be observed about distributions.

A somewhat inherent problem when we want to represent distributions explicitly is that they can quickly get big. For instance, if we want to find the sum of rolling a die three times:

```
die = uniform [1..6]
threeDice = (+) <$> die <*> ((+) <$> die <*> die)
```

we will build a `Dist` value with 6^3 elements (in this particular case we can normalise the distribution to only have 16 unique elements, but remember that we cannot always normalise distributions). However, what is more worrisome is that if we want to compute an expectation of a distribution or normalise it, then we have to fully evaluate the distribution and hold it in memory. For instance,

```
instance Monad Dist where
   return x    = D [(x, 1)]
   (D d) >>= f = D [(y,q*p) | (x,p) ← d
                            , (y,q) ← unD (f x)]

instance Functor Dist where
   fmap f (D d) = D [(f x,p) | (x,p) ← d]

instance Applicative Dist where
   pure = return
   df <*> dx = df >>= \f → dx >>= \x → return (f x)

choice :: Float → a → a → Dist a
choice p x y | 0.0 <= p && p <= 1.0 = D [(x, p), (y, 1-p)]

uniform :: [a] → Dist a
uniform xs = D [(x, 1 / total) | x ← xs]
  where total = fromIntegral (length xs)

type Valuation a = a → Float

expected :: Valuation a → Dist a → Float
expected value dist =
   sum $ map (\(x,p)→ p * value x) $ unD dist
```

Figure 1. Monadic library, with probability distributions represented as lists.

```
data Dist a where
  Certainly :: a → Dist a       -- only possible value
  Choice    :: Probability → Dist a → Dist a → Dist a
  Fmap      :: (a → b) → Dist a → Dist b
  Join      :: Dist(Dist a) → Dist a

certainly = Certainly

djoin :: Dist(Dist a) → Dist a
djoin ddist =
  case ddist of
    Certainly a    → a
    Choice p d1 d2 → Choice p (djoin d1) (djoin d2)
    _              → Join ddist

instance Functor Dist where
  fmap f (Certainly x) = Certainly $ f x
  fmap f (Fmap g d)    = Fmap (f . g) d
  fmap f (Join d)      = Join $ fmap (fmap f) d
  fmap f d             = Fmap f d

instance Applicative Dist where
  pure      = certainly
  df <*> dx = djoin $ fmap (flip fmap dx) df

instance Monad Dist where
   return  = certainly
   m >>= g = djoin $ fmap g m
```

Figure 2. Symbolic representation of distributions

just computing the sum of rolling a six-sided die eight times will take up nearly 2GB of RAM[1] if we don't use normalisation.

3. The Solution

The problem with the list-based representation in Figure 1 is that we are too eager to bring the representation to a kind-of normalised form. That is, we eagerly compute the probabilities for each outcome. What we should do instead is just postpone the computation of probabilities and make sure that we can traverse the distribution in a lazy fashion.

Figure 2 show a more symbolic representation of distributions where we use a generalised algebraic data type (GADT) with four constructors: `Certainly` x for the distribution with just one element, x; `Choice` p d_1 d_2 is the combined distribution of d_1 and d_2, where there is p probability of choosing an element from d_1; `Fmap` f d is the same distribution as d with f applied to all outcomes, that is all probabilities are unchanged; and `Join` d for flattening a distribution of distributions. Making this representation an instance of `Monad`, `Functor`, and `Applicative` is straight-forward and almost falls out naturally from the given types, we use the the monadic identities for making some slight optimisations in `fmap` and `djoin`, but they could just as well have been just the constructors `Fmap` and `Join`.

What is important to note, are the optimisations which are *not* made. For instance, while it is semantically unproblematic to let `fmap` distribute over `Choice`, doing so will be disastrous. Because that will destroy sharing, and more importantly it may introduce an exponentially number of new nodes. The given formulation of `fmap` will *at most* grow the height of distribution tree with one node, although it may also recreate an initial path of `Join` nodes. As we shall see in the following section, the bounded growth is an important property of `fmap`. Likewise, it is unproblematic to make a library without the `Join` and `Fmap` constructors, however this will inhibit the important property that `fmap` only grow the representation with one node, thus it can lead to huge values.

This representation still have the problem of non-canonical representation for a distribution. For instance, the following `Dist` values all represent the same distribution:

```
v1 = Choice 0.6 (Certainly 2) (Certainly 4)
v2 = Choice 0.5 (Certainly 2) (Choice 0.2 (Certainly 2)
                                          (Certainly 4))
v3 = Choice 0.4 (Certainly 4) (Choice 0.5 (Certainly 2)
                                          (Certainly 2))
```

and this just using `Choice` (and `Certainly`). Thus, we shall settle for the same kind of observable equivalence of distributions that we have for the list-based representation.

The interesting things to notice with this representation is that, when we construct new distributions we do not eagerly compute probabilities, and we are able to delay the flattening of distributions. Interestingly, we can apply a function to the leaves of a distribution of distribution, without flattening (joining) it first (see the function clause of `fmap` for `Join`).

3.1 Computing Expected Values

The symbolic representation allows us to compute expected results for a distribution in a compositional manner, where we lazily unfold the distribution:

```
expected :: Valuation a → Dist a → Float
expected v dexp =
  case dexp of
    Certainly a    → v a
    Choice p d1 d2 → p * (expected v d1)
                   + (1-p) * (expected v d2)
    Fmap f d       → expected (v . f) d
    Join d         → expected id (fmap (expected v) d)
```

Where the interesting parts are that we compute the values of the two distributions of `Choice` independently, and we that we only do some minimal bookkeeping in the traversal of `Fmap` and `Join` (recall that `fmap` will only introduce at most a single new `Fmap` node). Thus, we are be able to compute expected values in near constant space. The only unlimited space is stack-space in the recursive calls for `Choice`, however we could write `expected` to be tail-recursive by using an accumulating parameter. This is not done here for clarity of presentation.

3.2 Sampling Distributions

Similar to how we compute expected values, we can use the symbolic representation to write a sampling function that simulate sampling from the distribution. Figure 3 gives two definitions for sampling a distribution. The straightforward way, `sampleGen`, were we use a random number generator and draw one random number per `Choice` constructor. Another way to define a sampling function, `sample`, is to map random real numbers in the interval $[0; 1[$ to elements from the distribution, as done in [Ramsey and Pfeffer 2002]. In this definition we only need one number from the random number generator. However, it is no longer straightforward to argue that `sample` returns values from the distribution with the right probabilities, the proof is given in [Ramsey and Pfeffer 2002], however the proof relies on *real* real numbers with arbitrarily many (random) bits of precision.

3.3 Normalisation

If the elements of a distribution is ordered then we would like to be able to observe a normalised view of a distribution. That is, a mapping from the elements of the distribution to probabilities.

First we observe that the computation of expected values can be generalised, so that floating point numbers is not the only kind of expected value:

[1] Using GHC version 6.12.3 on Mac OS X (Snow Leopard).

```
sample :: Dist a → Float → (a, Float)
sample (Certainly a) r = (a, r)
sample (Choice p d1 d2) r =
                      if r < p then sample d1 (r/p)
                      else sample d2 ((1.0-r)/(1.0-p))
sample (Fmap f d) r = (f x, r')
                      where (x, r') = sample d r
sample (Join d) r   = sample d' r'
                      where (d', r') = sample d r

sampleGen :: R.RandomGen g => Dist a → g → (a, g)
sampleGen (Certainly a) g    = (a, g)
sampleGen (Choice p d1 d2) g =
                 sampleGen (if r < p then d1 else d2) g'
                 where (r, g') = R.random g
sampleGen (Fmap f d) g = (f x, g')
                         where (x, g') = sampleGen d g
sampleGen (Join d) g   = sampleGen d' g'
                         where (d', g') = sampleGen d g
```

Figure 3. Two defitions of sampling functions, one that only needs one random number and one that uses a random number generator.

```
expectedGen :: (Float → v → v) → (v → v → v)
            → (a → v) → Dist a → v
expectedGen scale comb val dexp =
  case dexp of
    Certainly a    → val a
    Choice p d1 d2 → (p `scale` (expected val d1))
                     `comb`
                     ((1-p) `scale` (expected val d2))
    Fmap f d       → expected (value . f) d
    Join d         → expected id $ fmap (expected val) d
  where
    expected = expectedGen scale comb
```

This is just the `expected` function from Section 3.1 parameterized with functions to scale a computed value, `scale`, to combine two values, `comb`, and with the valuation of elements, `val`, has a more general type.

Now, to compute a normalised view of a distribution we just need to supply suitable functions for `expectedGen`. For instance, we can compute a standard mapping:

```
normalise :: Ord a => Dist a → Map a Float
normalise = expectedGen scale comb value
  where
    value a = Map.singleton a 1.0
    scale p = Map.map (p *)
    comb    = Map.unionWith (+)
```

4. Evaluation

To test evaluate the library using the symbolic representation, from Figure 2, against the library using the list-based representation, from Figure 1, we have made two simple benchmarks: one based on the sum of rolling some dice, and one based on find the probability of getting the hand flush in poker. All benchmark tests were performed on a lightly loaded MacBook Pro with a Intel Core i5 CPU and 8 GB of RAM, using GHC version 6.12.3. To orchestrate the benchmarks we use the Haskell library criterion [O'Sullivan 2010], which automatically ensures that the benchmarks are iterated enough times to fit with the resolution of the clock. Furthermore, the criterion performs a bootstrap analysis on the timings to check that spikes in the load from other programs running on the computer do not skew the results. Each timing reported is the average of 50 samplings. Since the standard deviations of these averages are all negligible, we do not report the standard deviations.

4.1 Roll The Dice

To model that we roll a die with d sides n time we define the functions `die` and `rolls`:

```
die d = [1..d]

rolls d 1 = die d
rolls d n = (+) <$> die d <*> rolls d (n-1)
```

Following are the results of finding the expected sum of rolling a die with 5, 6, and 7 sides eight times:

No. of sides	5	6	7
Symbolic	121	500	1663
List	572	4351	35214

All times are given in milliseconds. Casual glance at an activity monitor confirm that the symbolic representation only uses a constant amount of memory, whereas the list-based represtation ends up using 1.5 GB for the final test. As it can be seen form the timings, the symbolic library still ends up traversing all 7^8 possible outcomes.

The example is somewhat contrived as we can drastically cut down the number of needed computations if we normalise the distributions as we go along. However, this holds equally for both representations.

4.2 Flush

A more interesting thing than rolling dice to model, might be the probability of getting a certain hand in poker. For instance, a flush (i.e., a hand where all cards have the same colour).

To model the drawing of an extra card and adding it to your hand we define the function `draw`:

```
draw cards = do
  card ← uniform (newDeck \\ cards)
  return (card:cards)
```

where `cards` is your current hand represented as a list, and `newDeck` is a complete deck of cards. The full code used for the modelling can be found in appendix A.

Following are the timings to find the expected chance of getting a flush when drawing 3, 4, and 5 cards:

No. of cards	3	4	5
Symbolic	40	2547	152000
List	40	2562	—

All times are given in milliseconds. Casual glance at an activity monitor during the benchmarking confirms that the symbolic representation only uses a constant amount of memory, whereas the list-based representation ends up using too much memory to complete the final test.

5. Related Work

Using a list-based representation for distributions have been suggested and described by [Filinski 1996], [Erwig and Kollmansberger 2006], and [Kidd 2007]. Where [Kidd 2007] builds the probability monad from a toolkit of monad transformers.

The idea of using a `Choice` operator and a non-normalised form comes from Troll [Mogensen 2009]. However, the combinators for building distributions in the Troll implementation is specialised for computing probabilities of rolling dice, that is distributions of multisets of integers. And the combinators are hard to generalise for arbitrary types.

In [Ramsey and Pfeffer 2002] one of the core constructors of probability monad is also `choice`. However, it is not clear is the

Haskell based implementation described in that paper also uses a list-based representation. To compute expectations efficiently [Ramsey and Pfeffer 2002] introduces *measure terms* that are somewhat related to the symbolic representation presented in this paper. My guess is that it might be possible to compute a measure term from the symbolic representation from Section 3, optimise the measure term, and then compute the expectation from the optimised measure term. However, I haven't tried that yet.

The idea of lazily unfolding the representation of distribution can also be found in [Kiselyov and Shan 2009], who uses lazy probabilistic search tree as one of their probability monads.

6. Concluding Remarks

Working with probabilistic models as a monadic library is convenient, but if it is not also efficient the charm wears off when trying to model even moderate examples. In the paper I have presented a symbolic representation which is almost as elegant as the list-based representation, and allows us a number of optimisations, for instance to compute expected result in constant space. However, time-wise we are no better of asymptotically than the list-based representation.

Using the symbolic representation for simulations is as easy as using the list-based representation, and have thus not been treated in this paper.

The symbolic representation might open up for more optimisation which are yet to be exploited. For instance, it would be most interesting to try and port over the implementation of *variable elimination* [Dechter 1998] from [Kiselyov and Shan 2009].

Acknowledgments

Thanks to Fritz Henglein, Andrzej Filinski, and Torben Mogensen for many fruitful discussions about monads, symbolic representation and optimizations, and probability distributions. The current article is much better because of these discussions. Also, thanks to Michael Werk and Joakim Ahnfelt for the poker example.

References

R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In M. I. Jordan, editor, *Learning and inference in graphical models*. MIT Press, 1998.

M. Erwig and S. Kollmansberger. Probabilistic functional programming in haskell. *Journal of Functional Programming*, 16(1):21–34, 2006.

A. Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1996.

E. Kidd. Build your own probability monads. In *Draft paper for Hac 07 II in Freiburg*, 2007.

O. Kiselyov and C.-c. Shan. Embedded probabilistic programming. In W. Taha, editor, *Proceedings of the IFIP working conference on domain-specific languages*, number 5658 in LNCS, pages 360–384. Springer, 2009.

T. Mogensen. Troll, a language for specifying dice-rolls. In *SAC09*, pages 1910–1915, March 2009.

B. O'Sullivan. Criterion package. Available from `http://hackage.haskell.org/package/criterion`, 2010.

N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *29th ACM POPL*, pages 154–165. ACM Press, 2002.

A. Poker Hands

The code for modelling poker hands with the library:

```
data Color = Clubs | Spades | Diamonds | Hearts
   deriving (Show, Ord, Eq, Enum)
type Card = (Int, Color)
type Deck = [Card]

newDeck :: Deck
newDeck = [(number, color) | number ← [1..13]
                           , color ← [Clubs .. Hearts]]

type Trans a = a → Dist a

draw :: Trans [Card]
draw cards = do
  card ← uniform $ newDeck \\ cards
  return (card:cards)

(*.) :: Int → Trans a → Trans a
1 *. t = t
n *. t = \x → ((n - 1) *. t) x >>= t

draws :: Int → Trans [Card]
draws n = n *. draw

hand :: Int → Dist [Card]
hand n = draws n []

isFlush :: [Card] → Bool
isFlush hand = length (nub (map snd hand)) == 1

flush n = expected (\v → if (isFlush v) then 1 else 0)
                   (hand n)
```

B. Full Code Listing

Figure 4 gives the full code listing.

```
data Dist a where
  Certainly :: a → Dist a       -- only possible value
  Choice    :: Probability → Dist a → Dist a → Dist a
  Fmap      :: (a → b) → Dist a → Dist b
  Join      :: Dist(Dist a) → Dist a

certainly = Certainly

choice :: Probability → Dist a → Dist a → Dist a
choice 1.0 d1 _ = d1
choice 0.0 _ d2 = d2
choice p d1 d2  = Choice p d1 d2

instance Functor Dist where
  fmap f (Certainly x) = Certainly $ f x
  fmap f (Fmap g d)    = Fmap (f . g) d
  fmap f (Join d)      = Join $ fmap (fmap f) d
  fmap f d             = Fmap f d

instance Applicative Dist where
  pure = certainly
  df <*> dx = djoin $ fmap (flip fmap dx) d
  _ *> dy = dy
  dx <* _ = dx

djoin :: Dist(Dist a) → Dist a
djoin ddist =
  case ddist of
    Certainly a    → a
    Choice p d1 d2 → Choice p (djoin d1) (djoin d2)
    _              → Join ddist

instance Monad Dist where
   return = certainly
   m >>= g = djoin $ fmap g m

uniform :: [a] → Dist a
uniform inp = choices 0.0 inp
  where total = fromIntegral $ length inp
        choices _ [e]      = certainly e
        choices n (e:rest) = choice (1/(total-n)) (certainly e) (choices (n+1) rest)

type Valuation a = a → Float

expected :: Valuation a → Dist a → Float
expected value dexp =
  case dexp of
    Certainly a    → value a
    Choice p d1 d2 → p * (expected value d1) + (1-p) * (expected value d2)
    Fmap f d       → expected (value . f) d
    Join d         → expected id $ fmap (expected value) d

sample :: Dist a → Float → (a, Float)
sample (Certainly a) r    = (a, r)
sample (Choice p d1 d2) r = if r < p then sample d1 (r/p)
                            else sample d2 ((1.0-r)/(1.0-p))
sample (Fmap f d) r       = (f x, r')
                            where (x, r') = sample d r
sample (Join d) r         = sample d' r'
                            where (d', r') = sample d r

expectedGen :: (Float → v → v) → (v → v → v) → (a → v) → Dist a → v
expectedGen scale comb value dexp =
  case dexp of
    Certainly a    → value a
    Choice p d1 d2 → (p `scale` (expected value d1))
                     `comb`
                     ((1-p) `scale` (expected value d2))
    Fmap f d       → expected (value . f) d
    Join d         → expected id $ fmap (expected value) d
  where
    expected = expectedGen scale comb

normalise :: Ord a => Dist a → Map a Float
normalise = expectedGen scale comb value
  where
    value a = Map.singleton a 1.0
    scale p = Map.map (p *)
    comb    = Map.unionWith (+)
```


Figure 4. Full code listing