All You Need is the Monad… What Monad Was That Again?

Norman Ramsey
Tufts University
nr@cs.tufts.edu

1. Introduction

Probability enjoys a monadic structure (Lawvere 1962; Giry 1981; Ramsey and Pfeffer 2002). A monadic computation represents a probability distribution, and the unit operation `return a` creates the (Dirac) distribution "certainly `a`." The bind operation combines a distribution of type `M a` and a function of type `a -> M b`; the function is a *probability kernel* (Pollard 2002), and it represents the conditional probability of `b` given `a`. The bind operation produces a new distribution which is defined using a Lebesgue integral, also called an "abstract" integral, over all possible values of type `a`. The Lebesgue integral works out in the same way whether the measurable space of `a`'s is discrete, continuous, or hybrid.

To write interesting distributions, one must include in one's monad one or more probabilistic operations, like discrete choice or a primitive distribution. Useful primitive distributions include a biased coin (Borgström et al. 2013), the uniform distribution over the unit interval (Park, Pfenning, and Thrun 2008), normal, beta, gamma, and other distributions.

Probability monads inform the design of several programming languages, including some cited above. But a probabilistic programming language must do more than just represent probability distributions; it must support *inference*. In Bayesian inference, a term in the language denotes a prior distribution, some information is observed, and using the prior distribution and the evidence provided by the observation, we calculate or estimate a posterior distribution. This operation is sometimes called *conditioning*.

Inference and conditioning raise design questions that appear not to have canonical answers. Here are some:

- What kinds of measures should be used to define the denotations of terms of monadic type?
- What is an observation? Is it a predicate or set? An index into a family of posterior distributions? A likelihood function?
- Where do we put fundamental operations, like choice and conditioning? "Outside," as functions that consume and transform monadic computations? Or "inside," as functions that return monadic computations but consume only pure values?

Even more interesting are questions about design tradeoffs and other properties of designs:

- How do our design choices affect the properties of the monad? In particular, how do they affect equational reasoning?
- Can we explain *why* certain semantic structures behave as they do? For example, why does extending the probability monad with conditioning go so badly wrong (Appendix A)?
- What guarantees can we provide to a programmer, and what problems is it up to the programmer to avoid? What design principles will help programmers avoid known semantic problems?

Some answers can be found in corners of various papers; for example, in settings that support inference, we shouldn't limit ourselves to probability measures. But the available answers don't yet add up to a coherent understanding of the design space. That coherent understanding, expressed in a single intellectual framework, is what I'm working toward.

2. Foundations

I want to support a typical agenda of functional programmers: by appealing to underlying semantics, I hope to justify algebraic laws that we can then use to transform programs. Semantics can give programs multiple interpretations. For example, in addition to a pencil-and-paper interpretation in measure theory, Ramsey and Pfeffer (2002) present Haskell interpretations for computing expectations, for sampling, and for computing support of finite distributions. Park, Pfenning, and Thrun (2008) develop the sampling interpretation into a full language with an impressive application in robotics. Another sampling interpretation, which I have not seen discussed in the programming-language literature, produces *weighted* samples for use in, e.g., importance sampling.

The foundational interpretation is measure-theoretic (Kolmogorov 1933; Pollard 2002): a term in the probability monad denotes a probability measure. A measure may be expressed as a *measure function*, which maps each measurable set to a nonnegative real number, or as an *integrator*, which integrates any measurable real-valued function and thereby maps it to an extended real number. The two formulations are equivalent: the measure function is the integrator of the characteristic function, and the integrator is the limit of a monotone sequence of linear combinations of applications of the measure function. But for language design and specification, the integrator is the more convenient foundation.

Signature algebraic law: commutativity What laws apply to integrators? The monad laws provide associativity and an identity (`return`). And for probability, we want commutativity. In particular, if m_1 and m_2 are independent of `x` and `y`, then the following terms, expressed in Haskell `do`-notation, ought to denote the same probability distribution:

```
do x <- m₁              do y <- m₂
   y <- m₂                 x <- m₁
   return (k x y)          return (k x y)
```

These two terms should denote the same distribution, but not the same sampler: they denote different samplers that both estimate the same underlying distribution. In an application, different samplers that estimate the same distribution are equally valid semantically, and commutativity justifies many transformations, including dead-binding elimination, which we wish to apply to samplers in order to improve other properties, like performance.

Unfortunately, commutativity holds only under certain conditions. Commutation amounts to changing the order of integration; it can be justified by applying Fubini's theorem or Tonelli's theorem. The measurable spaces of $\mathtt{x}$ and $\mathtt{y}$ must be σ-finite, which is not a problem, but the absolute value of the function being integrated must have a finite integral, which is impossible to guarantee. To take an example from Wikipedia, the programmer who asks for the expected value of

$$\frac{x^2 - y^2}{(x^2 + y^2)^2},$$

where x and y are uniformly distributed over the unit interval, is in for a rude surprise: depending on the order of integration, iterated integrals produce either $+\pi/4$ or $-\pi/4$. Is a function like this up to the programmer to avoid? Or can we rule it out by static analysis?

3. Design: All you need is what monad again?

At the workshop, I hope to explore design alternatives for monadic semantics of probabilistic languages: what do we put in the monad, and what properties and laws do we get? I say "all you need is the monad" because in a few key papers, important operations like choice or conditioning are expressed as measure transformers or other functions that consume monadic computations. But if we wish, we can express them instead within the monad.[1] I ask "what monad again?" because there are many monads that can express probabilistic modeling and inference, and as noted in my Introduction, I want to understand the design space.

In two pages I can't even sketch the design space, but here are some ideas and alternatives that have caught my interest:

- The probability monad is great for programmers: every well-typed term denotes a probability distribution, and the relation "denotes the same distribution" is a congruence over the operations of the monad.
- We can extend the probability monad with a measure transformer for conditioning. When used judiciously, such a conditioning operator is very useful; for example, it's easy to code sequential Bayesian inference. But things can go badly wrong: when used injudiciously, the conditioning operator produces wrong answers, some of which aren't even probability distributions (Appendix A). And as shown by Giannakopoulos, Wand, and Cobb (2015), "denotes the same distribution" is no longer a congruence: compositions of equivalent terms aren't always equivalent, so equational reasoning is crippled.
- We can repair the damage by turning to analogous operations on some other collection of measures: sub-probability measures, finite measures, or σ-finite measures. Every well-typed term denotes a measure, and "denotes the same measure" is a congruence. And we can express conditioning entirely within the monad, without having to introduce an operation that consumes monadic computations. But there are new problems: some terms denote the zero measure, which does not represent any probability distribution. An infinite measure also does not represent a distribution. Terms denoting such measures are terms that it is up to the programmer to avoid.

  Programmers also have a new worry: different measures can denote the same probability distribution while still having different *weights*. The weight is the integral of the constant 1 function, or if you prefer, the measure of the whole space. Mixture models defined by weighted-choice operations behave sensibly only if the measures denoted by the terms being mixed have equal weights. When writing code, programmers must therefore be aware of weights. There are pitfalls here: if, like me, you have trained yourself to think in terms of the probability monad, switching to a monad of measures makes it all too easy to write down terms that look plausible and denote valid measures but that don't mean what you intended. (Also, as noted in appendix A, the need to track and adjust weights makes the resulting language less compositional than it appears.)

  I wonder about a design with *two* type constructors for probabilistic computation. One, call it "distribution," sits in the probability monad. It guarantees equal weights and can easily express the intent of a mixture model. The other, call it "measure," sits in the larger monad of measures. It can easily express the results of inference (on either a distribution or another measure). To convert a measure to a distribution, divide it by its weight.
- In probability theory, an observation is a measurable set. In code, such a set is typically represented by a (measurable) function from values to Booleans. But this view doesn't enable useful inferences to be drawn from an observation of a set of measure zero—such as observing the value of a continuous variable. Shan and Ramsey (2015) propose expressing observation by writing a *term* giving the quantity to be observed, then using *disintegration* (Chang and Pollard 1997) to calculate a function from an observed value to a posterior distribution. Ścibior, Ghahramani, and Gordon (2015) propose expressing an observation by a function from values to *probabilities*—a *likelihood* function. I'd like to know how these choices relate and whether they can be unified under a single theory. (Preliminary result: it looks like the classic observation can be formulated as a special case of disintegration.)
- An infinite measure does not denote a probability distribution, so a designer might be tempted to rule them out. But there is good reason to allow them! For example, if we have a term denoting the Lebesgue measure, then every measure we call "continuous" is absolutely continuous with respect to the Lebesgue measure, and therefore it has a Radon-Nikodym derivative—a *probability density function*. The same reasoning applies to discrete spaces with respect to the (infinite) counting measure. And density functions, both continuous and discrete, are interesting and useful; for example, Riedel et al. (2014) describe WOLFE, an embedded probabilistic programming language based on manipulating density functions associated with explicitly identified measure spaces. I'd like to explore the algebraic structure of such a language.
- Many applications use sampling. Park, Pfenning, and Thrun (2008) use a monad like the one presented by Ramsey and Pfeffer (2002), which produces one sample at a time. But one can sometimes represent an underlying distribution more accurately, using fewer samples, if each sample is *weighted*. Weighted samples are especially natural in a monad that includes all finite measures, where reweighting or scaling a measure is likely to be included as an explicit operation. I'd like to know if making weights explicit, or even using (infinite) sets of weighted samples to denote distributions, enables interesting reasoning principles or program transformations.

A few of these ideas are developed in more detail in appendix A.

Acknowledgments

Theo Giannakopoulos helped sharpen many points that were vague. Sean Stromsten broadened my horizons. This work was supported by DARPA contract FA8750-14-C-0002.

References

Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. 2013. Measure transformer semantics for

[1] By "an operation within the monad" I mean a function that takes one or more pure values as arguments and returns a single monadic computation.

Bayesian machine learning. *Logical Methods in Computer Science*, 9 (3:11):1–39.

Joseph T. Chang and David Pollard. 1997. Conditioning as disintegration. *Statistica Neerlandica*, 51(3):287–317.

Theophilos Giannakopoulos, Mitchell Wand, and Andrew Cobb. 2015 (October). Finite-depth higher-order abstract syntax trees for reasoning about probabilistic programs. Manuscript to be submitted to the 2016 Workshop on Probabilistic Programming Semantics.

Michèle Giry. 1981. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes In Mathematics*, pages 68–85. Springer Verlag.

Andrey N. Kolmogorov. 1933. *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Second English edition, published in 1956 as *Foundations of the Theory of Probability*, Chelsea Publishing, New York, N.Y. Translation edited by Nathan Morrison. Library of Congress card catalog number 56-11512.

F. William Lawvere. 1962. The category of probabilistic mappings. Unpublished.

Sungwoo Park, Frank Pfenning, and Sebastian Thrun. 2008 (December). A probabilistic language based on sampling functions. *ACM Transactions on Programming Languages and Systems*, 31(1):4:1–4:46.

David Pollard. 2002. *A User's Guide to Measure Theoretic Probability*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press.

Norman Ramsey and Avi Pfeffer. 2002 (January). Stochastic lambda calculus and monads of probability distributions. *Proceedings of the 29th ACM Symposium on the Principles of Programming Languages,* in *SIGPLAN Notices*, 37(1):154–165.

Sebastian Riedel, Sameer Singh, Vivek Srikumar, Tim Rocktaschel, Larysa Visengeriyeva, and Jan Noessner. 2014. WOLFE: Strength reduction and approximate programming for probabilistic programming. In *International Workshop on Statistical Relational AI (StarAI)*, pages 100–103.

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

Chung-Chieh Shan and Norman Ramsey. 2015 (July). Symbolic Bayesian inference by lazy partial evaluation. Unpublished. See `http://www.cs.tufts.edu/~nr/pubs/disintegrator-abstract.html`.

A. A few ideas in more depth

Cutting back on measure transformers: choice Here are some variations on monads for probabilistic modeling and inference. One source of variation is a mistake I've made repeatedly: to overlook the power of monadic bind. For example, in my 2002 paper with Avi Pfeffer, we wrote discrete choice as a function that consumes monadic computations, which Borgström et al. (2013) call a "measure transformer":

```
class Monad m => ProbabilityMonad m where
  choose :: Probability -> m a -> m a -> m a
```

But we don't need a transformer; it's enough to provide a biased coin:

```
  class Monad m => ProbabilityMonad m where
    bernoulli :: Probability -> m Bool
```

The two are equivalent:

```
choose p m1 m2 = do first <- bernoulli p
                    if first then m1 else m2
bournoulli p =
  choose p (return True) (return False)
```

The relationship can also be seen in the expectation laws for `choose` and `bournoulli`:

```
expectation (choose p m1 m2) f =
                     p     * expectation m1 f +
                     (1-p) * expectation m2 f
expectation (bournoulli p) f =
                     p * f True + (1-p) * f False
```

These days I prefer `bournoulli`: the expectation law is simpler, and it needn't push me toward a design in which distributions are first-class objects.

Inference and monads In a monadic framework, how should we express inference or conditioning? If possible, we'd like to retain two pleasant properties of the probability monad *without* inference:

- The probability monad is *closed*: any well-typed monadic term denotes a probability distribution.[2] Closure makes the programmer's life easy: to know that he or she is working with probability distributions, the programmer relies on a static type system.
- The equivalence relation on terms, i.e., denoting the same probability distribution, is a *congruence*: compositions of equivalent terms are equivalent. Congruence expands the scope and power of equational reasoning, enabling the programmer to substitute one equivalent term for another in any context.

Unfortunately, when we add a conditioning operation to the probability monad, we lose both these properties. I suspect that the difficulties are well known—I know of no language based on a probability monad with conditioning—but they are not highlighted in the literature.

Here's one possible primitive for conditioning; it takes an observation and a prior distribution and returns a posterior distribution:

```
class ProbabilityMonad m => ConditioningMonad m where
  condition :: (a -> Bool) -> m a -> m a
```

To make measure-theoretic sense, an observation must be a measurable set; here the set is represented by its characteristic function, a measurable function from `a` to `Bool`.

In a simple implementation of finite probability by exhaustive enumeration, this primitive can go horribly wrong. For example, suppose you have a bowl of D&D dice: 4-sided, 6-sided, and so on up to 20-sided. From a hat you draw one of five slips of paper labeled with the numbers 1 to 5. You take that many dice from the bowl at random, throw them, and observe a total of 28. If the original slips were distributed uniformly, what's your posterior estimate of the number on the slip that was drawn?[3] If you code this problem carefully, you can get the right answer with the `condition` primitive. But you can also write reasonable-looking terms involving joint probability distributions and wind up with a totally wrong answer: slips 2 to 5 each have a probability of 20%. This answer is not only wrong—it isn't even a probability distribution! In addition, Giannakopoulos, Wand, and Cobb (2015) have shown that in a language with a very similar conditioning primitive, the equivalence "denotes the same probability distribution" is not a congruence.

One fix, which is also well known, is to move to a more general monad of measures. This might be a monad of sub-probability measures, of finite measures, or of general measures. The issues are probably all similar, but the monad of general measures is the most interesting. Why? Because only by including non-finite

[2] I have verified this property only for the finite, discrete probability in my 2002 paper with Avi Pfeffer, but I believe it holds for at least some continuous cases as well.

[3] Larger numbers are more likely; you have a 38% chance of having drawn the slip labeled 5, down to a 5% chance of having drawn slip 2. Slip 1 is impossible; you can't throw 28 with just one die.

measures can we include the Lebesgue measure on real spaces—and that's important because every absolutely continuous measure can be represented by integrating using the Lebesgue measure and a probability-density function.

I believe the monad of measures is closed and that the equivalence "denote the same measure" is a congruence—a belief not yet bolstered by proof. Let's see what the monad looks like.

In the monad of measures, the analog of expectation is integration, and the analog of observation is restriction:

```
class MonadPlus m => Measure m where
  integrate :: m a -> (a -> Real) -> Real
  restrict  :: (a -> Bool) -> m a -> m a
  ... more coming ...
```

These operations obey a useful algebraic law:

```
  integrate (restrict ok m) f =
      integrate (char ok /*/ f)
    where char ok a   = if ok a then 1 else 0
          (f /*/ g) a = f a * g a
```

Expectation is defined only when the measure in question has nonzero, finite weight:

```
  expectation m f =
          integrate m f / integrate m (const 1)
```

The monad of measures has a richer algebraic structure than the monad of probability measures. A linear combination of measures (with nonnegative coefficients) is also a measure, and it is useful to talk about the zero measure:

```
  scale     :: Real+ -> m a -> m a
  add       :: m a -> m a -> m a
  zero      :: m a
```

Like any operation that defines a measure, these operations are fully specified by saying how they affect integration:

```
  integrate (scale c m) f = c * integrate m f
  integrate (m1 `add` m2) f =
                  integrate m1 f + integrate m2 f
  integrate zero f = 0
```

Now that we are no longer limited to probability measures, we can express conditioning without having to take a monadic computation as input—the monad is all we need:

```
  observe :: Bool -> m ()
```

The `observe` primitive is specified by simple laws, and it can replace `restrict`:

```
  integrate (observe True)  f = f ()
  integrate (observe False) f = 0
  restrict ok m =
      do a <- m; observe (ok a); return a
```

Weight, choice, and loss of compositionality The laws relating `integrate`, `scale`, and `add` together can express the laws for probabilistic choice. In particular, we can eliminate `choose` (or `bernoulli`) and instead write

```
  choose p m1 m2 = scale p m1 `add` scale (1-p) m2
```

The validity of this derivation of `choose`, along with the validity of other choice primitives like "45% green, 50% red, and 5% yellow," relies on measures `m1` and `m2` having equal weight. In the probability monad, this weight is always one, but in a monad of measures, it doesn't matter what the weight is, as long as it's the same for each measure being combined. I see a tension here, the result of which is that the language is less compositional than it appears:

- Choice primitives, or just linear combinations of measures, are always well defined, but they express the programmer's intent only when the measures being combined are of equal weight.
- The observation primitives `restrict` and `observe`, unlike the conditioning operation in the probability monad, never knock you out of the monad or violate congruence, but the very essence of the way they work is that they change a measure's weight.

In principle, you can combine choice with inference in any way you like, and sure enough the result is always a valid measure—maybe even a nonzero measure. But in practice, not every valid combination means the right thing; if you want to write such combinations and understand there meanings, most likely you are going to have to insert operations that adjust weights. Once again, the problem is that "denotes the same distribution" is not a congruence.

What is an observation? In the examples above, as in most work in the field, an observation is made of a measurable set of type `a`, which is often represented as a function from `a` to Boolean. In this setting it is difficult to find a principled way to deal with observations whose prior probability is zero. Shan and Ramsey (2015) use *disintegration* (Chang and Pollard 1997) to extend a probabilistic model with a quantity that is intended to be observed. The observation is a value of type `a`, and a "disintegrator" maps every observation onto a new measure, which typically measures a space with fewer dimensions than the original model.

Another possibility, described by Ścibior, Ghahramani, and Gordon (2015), is to redefine observation not as a function from `a` to Boolean but as a function from `a` to probability—that is, as a density function. This function is called a *likelihood*, and while it is new to me, the authors treat it as routine. And using it enables the authors to create samplers that compose in interesting ways. I look forward to working out a measure-theoretic explanation.