Build your own probability monads

Eric Kidd
Dartmouth College
eric.kidd@dartmouth.edu

Abstract

Probability is often counter-intuitive, and it always involves a great deal of math. This is unfortunate, because many applications in robotics and AI increasingly rely on probability theory. We introduce a modular toolkit for constructing probability monads, and show that it can be used for everything from discrete distributions to weighted particle filtering. This modular approach allows us to present a single, easy-to-use API for working with many kinds of probability distributions.

Our toolkit combines several existing components (the list monad, the $Rand$ monad, and the $MaybeT$ monad transformer), with a stripped down version of $WriterT\ Prob$, and a new monad for sequential Monte Carlo sampling. Using these components, we show that $MaybeT$ can be used to implement Bayes' theorem. We also show how to implement a monad for weighted particle filtering.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; G.3 [*Probability and Statistics*]

General Terms Probability, Monads

Keywords Bayesian filtering, particle filters

> *A very senior Microsoft developer who moved to Google told me that Google works and thinks at a higher level of abstraction than Microsoft. "Google uses Bayesian filtering the way Microsoft uses the if statement," he said.* -Joel Spolsky

1. Introduction

Probability is notoriously tricky and counter-intuitive. It's easy to ignore prior probabilities, confuse $P(A|B)$ with $P(B|A)$, or get lost while trying to generalize Bayes' theorem. But we encounter probabilities more than ever, thanks to recent trends in search algorithms, robotics and artificial intelligence.

To address these challenges, researchers have built several excellent programming languages based on probability distribution monads [30, 27, 8]. Some of these languages use random sampling; others compute exact results. But all of these languages are delightful tools—they make previously subtle problems intuitive and easy.

But these waters are deeper than a casual glance reveals. Many problems in probability theory can be expressed in terms of monads [18, 11, 14]. And if we examine a few such monads, certain repeated patterns become obvious. In fact, most probability distribution monads can be built from a small "toolkit" of monads and monad transformers. The same parts are shared by discrete distributions, random sampling monads, and even particle filters.

In general, the monads built from this toolkit make pleasant programming languages. For example, imagine that we have an influenza test with a 30% false-positive rate, and a 10% false-negative rate [2]. If we assume that 10% of the population has influenza, what is the probability that someone with a positive test result is actually infected? Instead of messing around with Bayes' theorem, we can simply write:

```
fluStatusGivenPositiveTest = do
  fluStatus  ← percentWithFlu 10
  testResult ← if fluStatus ≡ Flu
               then percentPositive 70
               else percentPositive 10
  guard (testResult ≡ Pos)
  return fluStatus
```

This function will return a probability distribution of 44% Flu and 56% $Healthy$. Because only a small portion of the population is actually infected, the false positives actually outnumber the true ones. This monad in this example is similar to other implementations of the discrete probability monad [8], with the addition of $guard$, which is used to do implicit Bayesian filtering.

We make the following contributions:

- We introduce a toolkit for constructing probability distribution monads (Section 3). The toolkit consists of three monads (the list monad, a Monte Carlo monad, and a sequential Monte Carlo monad), and two monad transformers ($PerhapsT$ and $MaybeT$). We use this toolkit to recreate an existing probability distribution monad (Sections 3.2 and 5).
- We implement Bayes' theorem using the $MaybeT$ monad transformer (Sections 3.3 and 6). $MaybeT$ allows us to discard possible outcomes, neatly encapsulating the notion of sampling with rejections.
- We develop several new monads for sequential Monte Carlo sampling, also known as particle filtering (Sections 3.4 and 7). These monads support both rejection filters (using $MaybeT$) and weighted filters (using $PerhapsT$).

2. Background

Probability theory poses many challenges for programming language designers. We must choose from a bewildering variety of representations and techniques. We must work around the limitations of the human intuition, which is notoriously bad at probability. And we must keep the math from obscuring the actual problems of interest. Solving all these problems at once is beyond the scope

of any existing programming language. At best, we can hope to find a sweet spot in the design space.

2.1 Choosing a representation

Probability theory offers a rich variety of problem-solving tools. At times, this variety is perhaps too rich. For example, probability distributions may be represented as discrete distributions [8], random sampling functions [30, 27, 8], measure terms [30], Kalman filters [31, 10], multi-hypothesis Kalman filters [10], or particle filters [10, 6]. Similarly, Bayes' theorem may be implemented using explicit calculations, the rejection method [30], importance sampling [30], weighted particle filters, or more sophisticated techniques.

Of course, each of these choices comes with tradeoffs. Discrete distributions offer exact answers, but they require exponential time to solve certain problems. Sampling functions can represent arbitrary distributions, but they can only calculate approximate distributions. Kalman filters are extremely efficient, but they treat all distributions as simple Guassians.

To further complicate matters, many of these techniques are traditionally described in ways that blur the underlying algebraic connections. For example, the various implementations of Bayes' theorem have quite a lot in common. But those similarities are hard to see if we compare the textbook formula for Bayes' theorem [31] with a description of weighted particle filters [10].

In an ideal world, we would be able consolidate as many of these techniques as possible under a single programming interface, making the algebraic connections obvious.

2.2 Coping with faulty intuitions

Probability is often counter-intuitive. Even physicians, who work with diagnostic tests on a daily basis, commonly make order-of-magnitude errors in interpretation. In an informal study by Eddy, most physicians concluded that a patient with an apparently benign breast mass but a positive mammogram had a 75% chance of cancer. The actual chance of cancer, as calculated by Bayes' theorm, was only 7.7% [7].

Other probability puzzles are similarly misleading. In a famous *Parade Magazine* article, Marilyn vos Savant described the "Monty Hall" problem, in which contestants must choose from several doors, one of which hides a prize [35]. Most of vos Savant's readers chose the wrong door, thanks to subtle ambiguities in the problem statement and the use of inappropriate heuristics [23].

To correctly answer a question about probability, we must first specify the details. In particular, we must know the starting conditions, the protocols followed by any agents in the puzzle, and the exact values we want to compute. This kind of precise specification is well-suited to a programming language.

2.3 Separating the problem from the math

Once we've decided how to represent a problem, and specified it precisely, we still need to do the math. But the math itself is frequently a barrier—instead of focusing on the actual problem, we often wind up trying to remember which version of Bayes' theorem applies in our particular case. Even worse, this preoccupation with formulas can blind us to simpler ways of looking at the problem.

Consider the influenza test in Section 1. We know that the test has a 30% false-positive rate and a 10% false-negative rate. If 10% of the population has influenza, we know that

$$P(I) = 0.1 \qquad P(+|I) = 0.7 \qquad P(+|\neg I) = 0.1$$

where $P(I)$ is the probability that a patient has influenza, $P(+|I)$ is the probability of a true positive, and $P(+|\neg I)$ is the probability of a false positive. We can plug these numbers into Bayes' theorem, and calculate $P(I|+)$, the probability that patient has influenza, given a positive test result.

$$P(I|+) = \frac{P(+|I)P(I)}{P(+|I)P(I) + P(+|\neg I)P(\neg I)} = \frac{7}{16}$$

But we can solve this problem more easily using a visual approach. Assuming we have 100 patients, we can separate them into 10 patients with influenza, and 90 healthy patients. If we then represent positive test results with "+", our population looks like:

Influenza	+	+	+	+	+	+	+	◦	◦	◦
Healthy	+	◦	◦	◦	◦	◦	◦	◦	◦	◦
	+	◦	◦	◦	◦	◦	◦	◦	◦	◦
	+	◦	◦	◦	◦	◦	◦	◦	◦	◦
	+	◦	◦	◦	◦	◦	◦	◦	◦	◦
	+	◦	◦	◦	◦	◦	◦	◦	◦	◦
	+	◦	◦	◦	◦	◦	◦	◦	◦	◦
	+	◦	◦	◦	◦	◦	◦	◦	◦	◦
	+	◦	◦	◦	◦	◦	◦	◦	◦	◦
	+	◦	◦	◦	◦	◦	◦	◦	◦	◦

We can see that $7/16$ths of positive test results occur in patients with influenza. The ease with which we can read answers off this diagram represents our ideal; any programming language should be as straightforward.

$[\,]$	Lists of outcomes
$PerhapsT\ [\,]$	Discrete distributions
$MaybeT\ (PerhapsT\ [\,])$	. . . with rejection
MC	Monte Carlo sampling
$MaybeT\ MC$	. . . with rejection
$PerhapsT\ MC$	. . . with weights
SMC	Sequential Monte Carlo sampling
$MaybeT\ SMC$	. . . with rejection
$PerhapsT\ SMC$	. . . with weights

Table 1. The probability monad toolkit.

3. The probability monad toolkit

We are now ready to introduce our toolkit for building probability monads. Our toolkit relies on two main ideas:

1. *Monads* allow us to split a computation into two parts: the main program, and the bookkeeping details [22, 36]. In the main program, we describe our problem in high-level terms, leaving out most of the math. But behind the scenes, a monad keeps track of the math for us, calculating probabilities and applying Bayes' theorem.
2. *Monad transformers* allow us to start with base monads, and layer on extra features as needed [21, 17, 34, 9]. Our base monads represent simple ideas: lists, sampling functions, and sets of particles. Everything else—including the probability calculations, the weights of the particles, and the implementation of Bayes' theorem—is supplied by one or more monad transformers.

Using only three base monads and two monad transformers, we are able to construct a wide range of probability distribution monads (Table 1).

[] (list monad)	$PerhapsT\ []$	$MaybeT\ (PerhapsT\ [])$
(Flu, Pos)	7% (Flu, Pos)	7% $Just\ (Flu, Pos)$
(Flu, Neg)	3% (Flu, Neg)	3% $Nothing$
$(Healthy, Pos)$	9% $(Healthy, Pos)$	9% $Just\ (Healthy, Pos)$
$(Healthy, Neg)$	81% $(Healthy, Neg)$	81% $Nothing$

Table 2. Building a monad in layers, with example data.

3.1 The list monad

The list monad allows us to combine elements from several lists, generating every possible outcome [36]. Continuing with our influenza example, we define two data types:

$$\begin{array}{l}\mathbf{data}\ Status = Flu \mid Healthy\\ \mathbf{data}\ Test\ \ = Pos \mid Neg\end{array}$$

Using Haskell's **do**-notation, we pick one item from each list and return the result. In this example, the $\leftarrow$ symbol should be read as "pick one item from."

$$\begin{array}{l}outcomes :: [(Status, Test)]\\ outcomes = \mathbf{do}\\ \quad status \leftarrow [Flu, Healthy]\\ \quad test\ \ \leftarrow [Pos, Neg]\\ \quad return\ (status, test)\end{array}$$

The type declaration $[(Status, Test)]$ is important, because it tells Haskell to interpret the **do**-body as a computation in the list monad. When we run this code, it will return a list of every possible outcome:

(Flu, Pos)	(Flu, Neg)
$(Healthy, Pos)$	$(Healthy, Neg)$

Whenever the list monad encounters $\leftarrow$, it makes *every* possible choice, backtracking as necessary. The list monad also appears in Haskell and other languages as a *list comprehension*, a special syntax for building lists [36]:

$$\begin{array}{l}[(status, test) \mid status \leftarrow [Flu, Healthy],\\ \qquad\qquad\quad test\ \ \leftarrow [Pos, Neg]]\end{array}$$

For more information on the list monad, see Section 4.1.

3.2 The PerhapsT monad transformer

By itself, the list monad has no way to keep track of probabilities. We can fix this using the $PerhapsT$ monad transformer.

$$\mathbf{type}\ DDist = PerhapsT\ []$$

$PerhapsT$ takes an existing monad, and attaches a probability to each value in the computation. The probabilities are tracked invisibly in the background, giving us a discrete probability distribution:

7% (Flu, Pos)	3% (Flu, Neg)
9% $(Healthy, Pos)$	81% $(Healthy, Neg)$

The code that generates this distribution is similar to our previous example. We replace the lists with calls to $weighted$, which constructs weighted distributions:

$$\begin{array}{l}weightedOutcomes :: DDist\ (Status, Test)\\ weightedOutcomes = \mathbf{do}\\ \quad status \leftarrow weighted\ [(10, Flu), (90, Healthy)]\\ \quad test\ \ \leftarrow\\ \quad\quad \mathbf{if}\ (status \equiv Flu)\\ \quad\quad\quad \mathbf{then}\ weighted\ [(70, Pos), (30, Neg)]\\ \quad\quad\quad \mathbf{else}\ \ weighted\ [(10, Pos), (90, Neg)]\\ \quad return\ (status, test)\end{array}$$

When run, $weightedOutcomes$ returns a list of possible results and their probabilities, as shown in the table above. Note that the final $DDist$ monad is equivalent to Erwig and Kollmansberger's $Dist$ monad [8]. For more information on the $PerhapsT$ monad transformer, see Section 5.

3.3 The MaybeT monad transformer

In Section 2.3, we implemented Bayes' theorem by counting up all the patients marked "+", and ignoring the rest. We can get a similar effect using the $MaybeT$ monad transformer:

$$\mathbf{type}\ BDDist = MaybeT\ DDist$$

$MaybeT$ takes an existing monad, and replaces each value of type a with either $Just\ a$ or $Nothing$ [1]. The $Nothing$ values represent failed "branches" in the computation, values of no interest to us. Filtering for $test \equiv Pos$, we get:

7% $Just\ (Flu, Pos)$	3% $Nothing$
9% $Just\ (Healthy, Pos)$	81% $Nothing$

At the end of the computation, we can discard all the $Nothing$ values, and scale the remaining percentages so that they sum to 100%. Compare this example to the diagram in Section 2.3. Both are implementations of Bayes' theorem using the rejection method and a normalization factor [27, 31].

The $filteredWeightedOutcomes$ function is identical to our earlier $weightedOutcomes$, except for the type declaration and the $guard$ function on the second-to-last line:

$$\begin{array}{l}filteredWeightedOutcomes :: BDDist\ (Status, Test)\\ filteredWeightedOutcomes = \mathbf{do}\\ \quad status \leftarrow ...\\ \quad test\ \ \leftarrow ...\\ \quad guard\ (test \equiv Pos)\\ \quad return\ (status, test)\end{array}$$

The $guard$ function checks to see if $test \equiv Pos$, and if not, replaces the current branch of the computation with $Nothing$.

See Table 2 for a step-by-step breakdown of how the $PerhapsT$ and $MaybeT$ monad transformers are used to construct $BDDist$. For more information on the $MaybeT$ monad transformer, see Section 6.

3.4 The MC and SMC monads

Monte Carlo algorithms rely on random sampling to compute approximate answers [20]. We implement random sampling using the MC monad, described under various names by previous researchers [30, 27, 8]. The example code for the MC monad is identical to $weightedOutcomes$, except for the type declaration:

$$\begin{array}{l}sampledOutcomes :: MC\ (Status, Test)\\ sampledOutcomes = ...\\ sample\ sampledOutcomes\ 10\end{array}$$

The $sample$ function runs our code repeatedly, collecting the results in a list.

Sequential Monte Carlo sampling, also known as *particle filtering*, differs from ordinary sampling in that all our samples pass

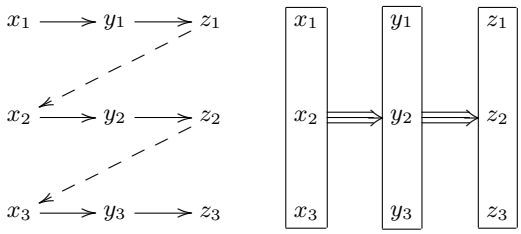

Figure 1. Monte Carlo sampling (left) and sequential Monte Carlo sampling (right). In the latter case, we represent our samples with a cloud of particles, which travel as a group through each step x, y, z of the computation.

through each step of the computation as a group [10, 6]. See Figure 3.4 for a comparison of the two approaches. We can perform sequential Monte Carlo sampling using the SMC monad.

The MC and SMC monads can also be combined with our monad transformers (Table 1). For example, we can use $PerhapsT$ to construct a weighted version of the SMC monad:

$$\textbf{type } WSMC = PerhapsT\ SMC$$

Using the $WSMC$ monad, we can replace our earlier calls to $guard$ with conditional probabilities. For example, if we know a patient's test result is positive, we can write:

```
statusGivenPosResult :: WSMC Status
statusGivenPosResult = do
  status    ← weighted [(10, Flu), (90, Healthy)]
  if (status ≡ Flu)
    then applyProb 0.7
    else applyProb 0.1
  return status
```

This is a classic weighted particle filter [10]. For more information on the MC and SMC monads, see Sections 4.3 and 7, respectively.

4. Monads and probability in Haskell

So far, our presentation of probability distribution monads has been informal. Now we provide the theory that supports the toolkit, beginning with a short introduction to monads and monad transformers.

4.1 Monads

In Haskell, a *type class* describes an abstract interface, which may be implemented by one or more actual types. The $Monad$ type class specifies two functions that must be defined by every monad m [36, 28]. It also constrains m to be a type constructor with a single argument:

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

The $return$ function takes a value of type a, and constructs a new value "in the monad," that is, a new value of type $m\ a$. The $\gg\!=$ operator[1] is a bit trickier. It is best understood as two operations, $liftM$ and $join$:

```
liftM  :: (Monad m) ⇒ (a → b) → m a → m b
join   :: (Monad m) ⇒ m (m a) → m a
```

[1] Read as "bind."

$$ma \gg\!= f = join\ (liftM\ f\ ma)$$

The $liftM$ function is analogous to the standard map function. Given a function of type $a \to b$, and value of type $m\ a$, it reaches inside $m\ a$, and replaces a with b. Similarly, the $join$ function is analogous to $concat$. It takes a nested value of type $m\ (m\ a)$, and collapses into a value of type $m\ a$.

For example, Haskell's standard list type forms a monad. Note the use of $concat$ and map in place of $join$ and $liftM$:

```
instance Monad [] where
  return x  = [x]
  ma >>= f  = concat (map f ma)
```

Haskell's **do**-notation is syntactic sugar for the $\gg\!=$ operator. It expands as follows:

```
do x ← [1, 2]
   return (x * 3)
```

$$[1,2] \gg\!= (\lambda x \to return\ (x * 3))$$

These two expressions are equivalent, and both return $[3, 6]$.

4.2 Probabilities and distributions

We represent a probability as a rational number, allowing us to work exact probabilities whenever possible.

```
newtype Prob = Prob Rational
  deriving (Eq, Ord, Num, Fractional)
```

The **deriving** clause automatically implements the specified type classes for us, based on the implementations for $Rational$.

The $Dist$ type class specifies the interface that must be implemented by a distribution type d.

```
class (Monad d) ⇒ Dist d where
  weighted :: [(Rational, a)] → d a
```

The constraint $(Monad\ d)$ requires all distributions to be monads. The $Dist$ type class requires only one function, $weighted$, which constructs a weighted distribution from a list of weights and values.

We can define a $uniform$ distribution by assigning each value a weight of 1:

```
uniform :: Dist d ⇒ [a] → d a
uniform = weighted ∘ map (λv → (1, v))
```

For a more realistic $Dist$ interface, see earlier papers by Park and colleagues [27] and Erwig and Kollmansberger [8].

4.3 The MC monad: An example

We can turn the proposed $Rand$ monad [1] into a probability distribution by defining a $Dist$ instance for it. This corresponds to the sampling monads which have appeared in a number of earlier papers [30, 27, 8].

```
type MC = Rand StdGen
instance Dist MC where
  weighted wvs = fromList (map flipPair wvs)
    where flipPair (a, b) = (b, a)
```

We create a type alias MC, which refers to $Rand\ StdGen$. The parameter $StdGen$ specifies what source of random numbers will be used, and the $fromList$ function makes a weighted selection from a list. We also define a $sample$ function:

```
sample :: MC a → Int → MC [a]
sample r n = sequence (replicate n r)
```

The standard $sequence$ function converts a value of type $[m\ a]$ into a value of type $m\ [a]$.

4.4 Monad transformers

In denotational semantics, a *monad morphism* is a mapping from one monad to another [21]. Monad morphisms take an underlying monad, and add features such as state, an environment, or continuations. Less formally, monad morphisms may be thought of as type-level functions from one monad to another.

In Haskell, we can achieve the same result using a monad transformer [19, 15, 33]. To define a monad transformer $ExampleT$, we must implement both $return$ and $\gg\!=$ in terms of m, the monad to be transformed:

$$\begin{aligned}&\textbf{instance}\ (Monad\ m) \Rightarrow Monad\ (ExampleT\ m)\ \textbf{where}\\&\quad return\ \ \ = ...\\&\quad ma \gg\!= f = ...\end{aligned}$$

We must also provide an implementation of *lift*, which maps values from a monad m into its transformed counterpart $t\ m$.

$$\begin{aligned}&\textbf{class}\ MonadTrans\ t\ \textbf{where}\\&\quad lift :: Monad\ m \Rightarrow m\ a \rightarrow t\ m\ a\end{aligned}$$

For examples of monad transformers, see Sections 5.3 and 6.

5. Monoids and discrete distributions

In this section, we introduce monoids, M-sets, and the MVT monad transformer. Using these pieces, we build the $DDist$ distribution described in Section 3.2. Special thanks go to Dan Piponi, for inspiring the treatment of M-sets in Section 5.2, and to Cale Gibbard, for first noticing the decomposition of $DDist$ into $WriterT\ Prob\ [\,]$.

5.1 Monoids

A *monoid* is a triple $(M, e, \otimes)$, where M is a set, $\otimes$ is an associative binary operation, and e is an identity element such that $e \otimes x = x \otimes e = x$. In Haskell, we can represent a monoid using the built-in $Monoid$ type class [15]:

$$\begin{aligned}&\textbf{class}\ Monoid\ a\ \textbf{where}\\&\quad mempty\ \ :: a\\&\quad mappend :: a \rightarrow a \rightarrow a\end{aligned}$$

Here, $mempty$ is e, and $mappend$ is $\otimes$.

Probabilities form a monoid $(P, 1, \times)$, where P is the set of all real numbers between 0 and 1 inclusive, and $\times$ is ordinary multiplication. In Haskell, we write this as:

$$\begin{aligned}&\textbf{instance}\ Monoid\ Prob\ \textbf{where}\\&\quad mempty\ \ \ \ \ \ = 1\\&\quad p_1\ `mappend`\ p_2 = p_1 * p_2\end{aligned}$$

5.2 M-sets and the MV monad

An *M-set* is a set X with a monoid action $(\cdot)$, such that

$$e \cdot x = x \tag{1}$$

$$m_1 \cdot (m_2 \cdot x) = (m_1 \otimes m_2) \cdot x \tag{2}$$

where $x \in X$, and $m, n \in M$. A *free M-set* is any M-set where

$$m \cdot x = x \qquad \text{only if } m = e. \tag{3}$$

For an equivalent definition of M-sets, see Kilp and colleagues [16, pp. 43–44,68].

Given a set S, define $P \times S$ to be the set of pairs (p, s) such that $p \in P$ and $s \in S$. In other words, the set $P \times S$ corresponds to elements of S annotated with probabilities. Now define our monoid action to be $p_1 \cdot (p_2, s) = (p_1 p_2, s)$. We can easily see that

$$1 \cdot (p, s) = (p, s) \tag{4}$$

$$\begin{aligned}p_1 \cdot (p_2 \cdot (p_3, s)) &= (p_1 p_2 p_3, s)\\&= (p_1 p_2) \cdot (p_3, s)\end{aligned} \tag{5}$$

$$p_1 \cdot (p_2, s) = (p_2, s) \qquad \text{only if } p_1 = 1. \tag{6}$$

From this, we conclude that $P \times S$ is a free M-set.

In Haskell, we can represent an element of an M-set using the type MV, which contains a monoid and a value.

$$\begin{aligned}&\textbf{data}\ (Monoid\ w) \Rightarrow MV\ w\ a =\\&\quad MV\{\,mvMonoid :: w, mvValue :: a\,\}\end{aligned}$$

The type $MV\ \ Prob\ \ a$ corresponds to the set $P \times S$ above. Interestingly, we can make MV a monad.

$$\begin{aligned}&mapMV\ f\ (MV\ w\ v) = MV\ w\ (f\ v)\\&joinMV\ (MV\ w_1\ (MV\ w_2\ v)) =\\&\quad MV\ (w_1\ `mappend`\ w_2)\ v\\&\textbf{instance}\ (Monoid\ w) \Rightarrow Monad\ (MV\ w)\ \textbf{where}\\&\quad return\ v = MV\ mempty\ v\\&\quad mv \gg\!= f = joinMV\ (mapMV\ f\ mv)\end{aligned}$$

The $return$ function corresponds to a map $x \mapsto (e, x)$, and $mempty$ to the monoid identity e. The $joinMV$ function corresponds to our monoid action $(\cdot)$.

Haskell programmers will recognize the MV monad as a stripped-down version of the standard $Writer$ monad [15]. We omit the *listen* and *pass* functions, which are useless in this context. We also omit the *tell* function, which we replace with $applyProb$:

$$\begin{aligned}&\textbf{class}\ (Monad\ m) \Rightarrow MonadCondProb\ m\ \textbf{where}\\&\quad applyProb :: Prob \rightarrow m\ ()\\&\textbf{instance}\ MonadCondProb\ (MV\ Prob)\ \textbf{where}\\&\quad applyProb\ p = MV\ p\ ()\end{aligned}$$

We can use $applyProb$ and MV to multiply a set of independent probabilities together. For example, if we have an influenza test with a 30% false negative rate, and we know that 10% of the population has influenza, we can calculate the actual percentage people receiving a false positive:

$$\begin{aligned}&flu\ \ \ \ \ \ \ \ \ \ \ \ = applyProb\ 0.1\\&falseNegative\ \ \ = applyProb\ 0.3\\&missedFluCases :: MV\ Prob\ ()\\&missedFluCases = \textbf{do}\\&\quad flu\\&\quad falseNegative\end{aligned}$$

The MV monad multiplies 0.1 and 0.3 behind the scenes, and returns $MV\ 0.03\ ()$. This tells us that 3% of the population will have influenza *and* incorrectly receive a negative test result. Conceptually, $missedFluCases$ corresponds to a single "path" through the example in Section 3.2, picking out only the case (Flu, Neg). Note that $MV\ \ Prob$ is not a probability distribution monad! At best, it represents a single element of a probability distribution.

5.3 The MVT monad transformer

The MV monad is only marginally useful by itself. We need some way to combine the features of MV with other monads. We do this by defining MVT, a stripped-down version of the $WriterT$ monad transformer [15].

$$\begin{aligned}&\textbf{newtype}\ (Monoid\ w, Monad\ m) \Rightarrow MVT\ w\ m\ a =\\&\quad MVT\{\,runMVT :: m\ (MV\ w\ a)\,\}\\&\textbf{instance}\ (Monoid\ w) \Rightarrow\end{aligned}$$

```
        MonadTrans (MVT w) where
  lift mv = MVT (do v ← mv
                    return (MV mempty v))
```

The trick is to take our underlying monad $m\ a$, and replace every occurrence of a with $MV\ w\ a$. The $runMVT$ function simply extracts the value packed inside our MVT wrapper. The *lift* function will take an existing value of type $m\ a$, and promotes it to a value of type $MVT\ w\ a$.

The $Monad$ instance for MVT is closely modelled on that of MV.

```
instance (Monoid w, Monad m) ⇒
         Monad (MVT w m) where
  return    = lift ∘ return
  ma ≫= f = MVT bound
    where bound = do
                  (MV w1 v1) ← runMVT ma
                  (MV w2 v2) ← runMVT (f v1)
                  return (MV (w1 `mappend` w2) v2)
```

We define the $return$ function for MVT in terms of *lift* and the underlying monad's $return$. The bind function is a bit trickier. We need $runMVT\ ma$ to get a value of type $m\ (MV\ w\ a)$, and use $\leftarrow$ to strip off m. The final two lines perform the same work as $mapMV$ and $bindMV$, respectively, but do so in the context of our underlying monad.

5.4 The DDist monad

Now we're finally ready to build our discrete distribution monad. We define $PerhapsT$ to be an alias for $MVT\ Prob$, and apply it to the standard list monad.

```
type PerhapsT = MVT Prob
type DDist = PerhapsT []
```

The list monad, as we saw in Section 3.1, follows every possible "branch" of a computation. When we apply $PerhapsT$ to the list monad, we associate a probability with each branch. To split a branch into sub-branches, we use $weighted$.

```
instance Dist DDist where
  weighted wvs = MVT (map toMV wvs)
    where toMV (w, v) = MV (Prob (w / total)) v
          total = sum (map fst wvs)
```

Note that we use $total$ to normalize the weights, forcing them to add up to 1. This not only ensures that $weighted$ returns a valid probability distribution, it also guarantees that the weights calculated by $\gg=$ for our sub-branches will add up to the weight of our original branch.

The $DDist$ monad is a stripped-down version of Erwig and Kollmansberger's probability monad [8]. The factoring of $DDist$ into $WriterT\ Prob\ []$ has been independently discovered by several people, including Cale Gibbard.

6. Bayes' theorem and MaybeT

In Sections 2.3 and 3.3, we used Bayes' theorem [5] to interpret the result of an influenza test. In this section, we show how to implement Bayes' theorem using the $MaybeT$ monad transformer.

6.1 Bayes' theorem

We adopt the convention that $\mathbf{P}(A)$ specifies a vector of probabilities, one for each possible value of A.

$$\mathbf{P}(A) = \langle P(A = a_1), \ldots, P(A = a_n) \rangle \tag{7}$$

Two such vectors may be multiplied pointwise.

$$\mathbf{P}(B = b|A)\mathbf{P}(A) = \langle P(B = b|A = a_1)P(A = a_1), \ldots, P(B = b|A = a_n)P(A = a_n) \rangle \tag{8}$$

Using this notation, we can state Bayes' theorem [31, p. 479] as

$$\mathbf{P}(A|B = b) = \frac{\mathbf{P}(B = b|A)\mathbf{P}(A)}{P(B = b)} \tag{9}$$

$$= \frac{\mathbf{P}(B = b|A)\mathbf{P}(A)}{\sum_i P(B = b|A = a_i)P(A = a_i)} \tag{10}$$

Now recall the program from Section 3.3.

```
filteredWeightedOutcomes :: BDDist (Status, Test)
filteredWeightedOutcomes = do
  status ← weighted [(10, Flu), (90, Healthy)]
  test   ← ...
  guard (test ≡ Pos)
  return (status, test)
```

This returns a list of possible results, and their corresponding probabilities:

```
7% Just (Flu, Pos)        3% Nothing
9% Just (Healthy, Pos)   81% Nothing
```

The $Nothing$ values represent those branches of our computation on which $guard\ (test \equiv Pos)$ failed.

We can apply Bayes' theorem to this example as follows. Let

$$\mathbf{P}(A) = \langle P(A = a_1), \ldots, P(A = a_n) \rangle \tag{11}$$

where $a_1, \ldots, a_n$ are the results of each branch of our computation. Let $G = g_1 \wedge \cdots \wedge g_m$, where g_i is true if and only if our i-th guard clause succeeds. This gives us

$$\mathbf{P}(A|G) = \frac{\mathbf{P}(G|A)\mathbf{P}(A)}{\sum_i P(G|A = a_i)P(A = a_i)} \tag{12}$$

But for each branch a_i of our computation, G is true if and only if $a_i \neq Nothing$. This gives us

$$P(G|A = a_i) = \begin{cases} 1 & \text{if } a_i \neq Nothing \\ 0 & \text{if } a_i = Nothing \end{cases} \tag{13}$$

allowing us to simplify (12) to

$$\mathbf{P}(A|G) = \frac{\mathbf{P}(G|A)\mathbf{P}(A)}{\sum_{a_i \neq Nothing} P(A = a_i)} \tag{14}$$

But this is equivalent to discarding all the $Nothing$ terms, and normalizing the remaining probabilities. Compare this result to the diagram in Section 2.3.

6.2 The MaybeT monad transformer

To build our Bayesian monad, we need an implementation of $MaybeT$ [1]. The $MaybeT$ monad transformer lifts a computation of type $m\ a$ to a computation of type $m\ (Maybe\ a)$.

```
newtype MaybeT m a =
  MaybeT { runMaybeT :: m (Maybe a) }
instance MonadTrans MaybeT where
  lift x = MaybeT (liftM Just x)
```

Here, $liftM\ Just$ has the type $m\ a \rightarrow m\ (Maybe\ a)$, where m is the monad being lifted.

We also need to define new versions of $return$ and $\gg=$ using the versions defined by m.

```
instance (Monad m) ⇒ Monad (MaybeT m) where
  return x  = lift (return x)
  ma ≫= f = MaybeT (runMaybeT ma ≫= f')
```

```
  where f' Nothing = return Nothing
        f' (Just x) = runMaybeT (f x)
```

There are two key concepts here:

1. Neither *return* or $\gg=$ create new *Nothing* values. This must be done using *mzero*, defined in Section 6.3.
2. Once a *Nothing* value is inserted into the monad, it will be passed along unchanged by $\gg=$, suppressing all further calls to any function f.

Note that we provide no *MonadPlus* instance for *MaybeT*, because it has ambiguous semantics. It could refer to either

```
instance (MonadPlus m) => MonadPlus (MaybeT m)
```

which lifts the semantics of an underlying *MonadPlus* instance, or

```
instance (Monad m) => MonadPlus (MaybeT m)
```

which provides semantics similar to *MonadPlus Maybe*. Therefore, we leave the choice up to the user of *MaybeT*.

6.3 The BDDist monad

Now we are ready to define *BDDist*, a discrete distribution monad with support for Bayes' theorem. We apply *MaybeT* to *DDist*, and lift the underlying *weighted* function into the new monad.

```
type BDDist = MaybeT DDist

instance (Dist d) => Dist (MaybeT d) where
  weighted wvs = lift (weighted wvs)
```

The *guard* function[2] is actually supplied by the standard Haskell library [28]:

```
guard :: MonadPlus m => Bool -> m ()
guard cond = if cond then return () else mzero
```

If *cond* is true, then *guard* returns (). This continues the current branch of the computation unchanged. But if *cond* is false, *guard* returns *mzero*. This has the effect of injecting *Nothing* into the computation, killing off the current branch.

To take advantage of *guard*, we need to make *BDDist* an instance of *MonadPlus*. Haskell also forces us to supply *mplus*, which we don't need in this paper.

```
instance MonadPlus BDDist where
  mzero = MaybeT (return Nothing)
  d1 `mplus` d2 = ...
```

We now have a monad which supports *guard*, and returns values of type *Maybe a*. We want to turn that *Maybe a* back into *a*, keeping the *Just* values and discarding the *Nothing* values. We can do this using *catMaybes'*.

```
catMaybes' :: (Monoid w) =>
              [MV w (Maybe a)] -> [MV w a]
catMaybes' = map (liftM fromJust) .
             filter (isJust . mvValue)
```

Now we're ready to implement Bayes' theorem, following the strategy outlined in Section 6.1. We use *catMaybes'* to discard all the *Nothing* values, and sum the remaining probabilities into *total*.

```
bayes :: BDDist a -> Maybe (DDist a)
bayes bfd
    | total == 0 = Nothing
    | otherwise = Just (weighted (map unpack events))
  where
    events = catMaybes' (runMVT (runMaybeT bfd))
    total  = sum (map mvMonoid events)
    unpack (MV (Prob p) v) = (p, v)
```

If *total* $\equiv$ 0, then our *guard* conditions have failed on every possible path, and we return *Nothing*. Otherwise, we construct a discrete probability distribution, using *weighted* to perform the actual normalization.

6.4 The BMC monad

We can use *MaybeT* to define a second Bayesian monad, this one based on rejection sampling [27]. Again, we omit the details of *mplus*.

```
type BMC = MaybeT MC

instance MonadPlus BMC where
  mzero = MaybeT (return Nothing)
  d1 `mplus` d2 = ...
```

The *BMC* monad returns samples of type *Maybe a*. Once again, we want to keep the *Just* values and discard the *Nothing* values. We can do that using the *sampleWithRejections* function, which takes n samples from a distribution, rejects those samples equal to *Nothing*, and returns the rest. These remaining samples are the ones that made it by all of our *guard* conditions.

```
sampleWithRejections :: BMC a -> Int -> MC [a]
sampleWithRejections d n =
  (liftM catMaybes) (sample (runMaybeT d) n)
```

Note that this function may return far fewer than n samples. This is because the distribution d, in a worst-case scenario, may never produce any samples at all. This will occur with the distribution *guard False*, and other distributions with impossible-to-satisfy guard conditions. Enhanced versions of *sampleWithRejections* must take care not to hang in these circumstances.

7. Sequential Monte Carlo sampling

Sequential Monte Carlo sampling is used in robot localization, computer vision, signal processing and econometrics [10, 6]. It differs from regular Monte Carlo sampling in that it represents samples as sets of "particles," the values of which are updated over time. In a typical application, each particle might represent one possible location of a robot. Initially, the particles are positioned at random. As the robot moves, the particles are moved along with it (perhaps with some random variation, if the speed of the robot is uncertain). If a particle winds up in an impossible location, such as inside a wall, that particle can be discarded and a new particle allocated elsewhere. The cloud of particles will eventually converge on one or more probable locations for the robot.

The major advantage of sequential Monte Carlo sampling over ordinary sampling is the ability to reallocate particles dynamically. This allows the algorithm to focus resources on the most interesting hypotheses, and therefore reduces the total number of samples required.

7.1 The SMC monad

We define our *SMC* monad in terms of *MC*. Internally, we represent *SMC* as a function mapping the desired number of samples to a list of actual samples. The list itself must be in the *MC* monad because it can only be generated using random numbers.

```
newtype SMC a =
  SMC{ runSMC :: Int -> MC [a] }

liftMC :: MC a -> SMC a
liftMC r = SMC (sample r)
```

[2] Earlier versions of this paper used a *condition* function. Thank you to David House for noticing that *condition* was identical to *guard*.

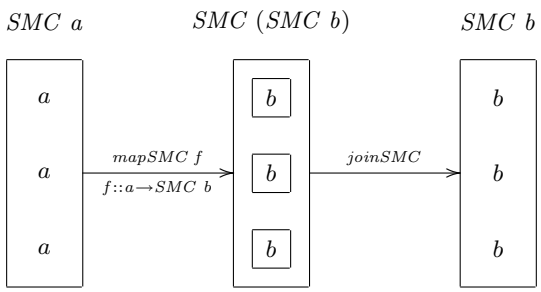

Figure 2. $mapSMC$ and $joinSMC$.

We can lift a distribution from MC to SMC simply by taking the requested number of samples. The $mapSMC$ function is a bit more complex, however. We define a new sampling function, $mapped$, which first turns a distribution d into actual samples, and then applies f to each sample. We package $mapped$ inside our SMC monad, where it can be run when needed.

```
mapSMC :: (a → b) → SMC a → SMC b
mapSMC f d = SMC mapped
  where mapped n = liftM (map f) (runSMC d n)
```

The $joinSMC$ function is the heart of the SMC monad. We begin with a cloud of particles, where each particle is itself an entire cloud (Figure 7.1, middle column). We want to take a single particle from each nested cloud, and put those particles into a new, flattened cloud.

To do this, we define a function $joined$, which takes n samples from the outermost cloud, and stores them in the list ds, which has type $[SMC\ a]$. It then takes one sample from each of these clouds using $sample1$, and stores them in xss, which has type $[[a]]$. All that remains is to flatten this list using $concat$.

```
joinSMC :: SMC (SMC a) → SMC a
joinSMC dd = SMC joined
  where joined n = do
    ds ← (runSMC dd n)
    xss ← sequence (map sample1 ds)
    return (concat xss)
    sample1 d = runSMC d 1
```

By taking only one sample from each of our nested clouds of particles, we effectively propagate each particle into one of its own possible futures.

The remaining code is trivial.

```
instance Monad SMC where
  return = liftMC ∘ return
  ps >>= f = joinSMC (mapSMC f ps)
instance Dist SMC where
  weighted = liftMC ∘ weighted
```

7.2 The WSMC monad and weighted particle filtering

Sequential Monte Carlo sampling is commonly performed using weighted particles. This is a form of *importance sampling*, where each sample has an associated weight [6]. A high-importance sample is interpreted as more "likely" than a low importance sample. This allows us to get more data out of a given number samples. Instead of discarding samples, as we did using $MaybeT$ in Section 6, we can simply mark those samples as unlikely.

The definition of the $WSMC$ monad should be unsurprising at this point. It relies on the same techniques seen in Section 6, with only cosmetic differences.

```
type WSMC = PerhapsT SMC
instance Dist WSMC where
  weighted wvs = lift (weighted wvs)
runWSMC :: WSMC a → Int → MC [MV Prob a]
runWSMC wps n = runSMC (runMVT wps) n
```

We do, however, provide two new features. The $applyProb$ function is a weighted version of $guard$. It multiplies the current particle's weight by p. Recall that p here is a probability, so we know that $0 \leq p \leq 1$. The actual multiplication is handled for us by $PerhapsT$.

```
instance MonadCondProb WSMC where
  applyProb p = MVT (return (MV p ()))
```

In practice, $applyProb$ is used to apply a new piece of evidence to our particles. For example, imagine that we have a robot with a door sensor. Let x be the current reading of the door sensor, and let $P(x|pos)$ be the probability of observing x at pos. We want to call $applyProb$ with an argument of $P(x|pos)$, which will apply an appropriate weight to each of our particles. For an excellent illustration of this process, see Fox and colleagues [10].

As time goes on, however, repeated calls to $applyProb$ will leave many of our particles with weights near zero. Periodically, we want to replace these particles with higher-probability ones. We can do this using the $resample$ function.

```
resample :: WSMC a → WSMC a
resample d = lift (SMC resampled)
  where resampled n = do
          xs ← runSMC (runMVT d) n
          sample (weighted (map unpack xs)) n
        unpack (MV (Prob p) x) = (p, x)
```

The $resample$ function treats our weighted particles as a probability distribution, and takes n new samples. This process will favor particles with high weights over particles with low weights, concentrating most of our particles in places where they'll do the most good. This particular implementation of resampling, however, is extremely naive and has statistical problems. See Doucet and colleagues for a survey of more sophisticated approaches [6].

8. Related work

Probability distribution monads are discussed by Lawvere [18] and Giry [11]. Giry's work focuses on Markov processes and transition probabilities, and provides a categorical foundation for probability measures. Jones and Plotkin lay further mathematical foundations, introducing a probabilistic power domain and showing that it forms a monad [14]. They also provide a λ-calculus model for the probability monad, and propose a probabilistic choice construct serving the same purpose as $weighted$.

Ramsey and Pfeffer provide Haskell interfaces for several probability distribution monads, including a sampling monad that corresponds roughly to MC [30]. They show that monads offer efficient implementations of support and sampling, but may be exponentially slower than other techniques for calculating expectations. To solve this problem, they translate the λ-calculus into measure terms.

Park and colleagues use a sampling monad in a variety of real-world robotics applications [27]. Their $\lambda_{\bigcirc}$-calculus uses sampling functions to represent binomial, geometric, and Gaussian distributions. They provide two implementations of Bayes' theorem

(one using the rejection method, the other using importance sampling), and a primitive for calculating expectations. This is the best-developed version of the MC monad in the literature.

Erwig and Kollmansberger have written an excellent Haskell library supporting both discrete distributions and random sampling, corresponding to the $DDist$ and MC monads in this paper [8]. They provide a much richer set of higher-order functions than we do, including functions for iterated simulation.

Our work differs from these earlier papers in several ways. We focus on the underlying structure of the probability monads, showing how to build them from layers of monad transformers. We also show that Bayes' theorem may be expressed naturally using $MaybeT$, removing the need for special primitives. And we introduce a new family of monads for sequential Monte Carlo sampling.

8.1 Monad morphisms and monad transformers

The history of monad morphisms and monad transformers has been described in a bibliography by Chung-chieh Shan [33]. Moggi originally used monad morphisms to build a modular theory of denotational semantics [21]. Moggi's theory was adapted for use in functional programming languages by King and Wadler [17]. Later work focused heavily on modular interpreters [37, 34, 9, 19, 13]. A typical modular interpreter is based on an $eval$ function in an unspecified monad. On top of this foundation, several layers of monad transformers are used to implement state, continuations, error reporting, and other language features.

In another field, Chung-chieh Shan applied monad morphisms to natural language semantics. He used various monad morphisms to construct a modular theory of interrogatives, focus, intensionality and quantification [32].

Monad transformers are well-supported by the Haskell programming language [15]. For good tutorials, see Grabmueller [12] and Newburn [25].

8.2 Other representations of probability distributions

A variety of techniques have been used to represent probability distributions. Fox and colleagues provide an excellent survey of Bayesian filtering and belief representations [10]. Park and colleagues also cover much of this ground in their paper on probability distribution monads [27].

In robotics applications, probability distributions are often represented as Kalman filters [31, 10]. Kalman filters are highly efficient, but they can only represent simple Gaussian distributions. Under certain assumptions, however, Kalman filters are theoretically optimal [10]. Unfortunately, our probability monad tookit does not support Kalman filters, because Gaussian distributions cannot be defined over arbitrary Haskell data types.

There is also an extensive literature on sequential Monte Carlo sampling and particle filters. For a good overview, see Fox and colleagues [10]. For a detailed discussion of current techniques, see Doucet and colleagues [6].

8.3 Probabilistic programming languages

Probabilistic programming languages are an enormously rich area of research. We describe a few here, just to give a sense of the sheer diversity.

Pfeffer's IBAL programming language supports sophisticated probabilistic inference [29]. IBAL is a functional language, with constructs similar to those in this paper. But where our monads directly calculate probability distributions, IBAL uses a two-phase inference algorithm to efficiently solve large problems. Phase 1 analyzes the program and produces an optimized computation graph. Phase 2 solves the computation graph and returns a probability distribution over the possible outputs.

Allison describes a Haskell library for machine learning and inductive inference [3, 4]. Inductive inference constructs a Bayesian network from real-world data, filling in the connections automatically. The major drawback of this approach is the risk of "overfitting," which occurs when the inference engine constructs a excessively-detailed model that describes every quirk of the training data. To avoid this problem, Allison uses Minimum Message Length (MML) to choose a model minimizing the combined size of the model and the compressed training data.

A rich variety of probabilistic logic programming languages have also appeared in the literature. Some examples include Ng and Subrahmanian [26], and Muggleton [24].

9. Conclusion

In this paper, we introduced a toolkit for building probability monads. Many components of the toolkit already existed in standard Haskell, or in various proposed libraries. These included the list monad, the Monte Carlo sampling monad, and the $MaybeT$ monad transformer. Two of our components, however, were new: The SMC monad (Section 7), which performs sequential Monte Carlo sampling, and the $PerhapsT$ monad transformer (Section 5), which is a stripped-down version of $WriterT\ Prob$.

We also demonstrated some novel applications of this toolkit. These included an implementation of Bayes' theorem using $MaybeT$ (Section 6), and a monad which performs weighted particle filtering (Section 7.2).

The most important benefit of the modular toolkit, however, has been the ease with which we can construct new monads. This facilitates tinkering and experimentation, and frequently offers us new perspectives on well-known techniques. Many of the monads in this paper were discovered by asking, "Hey, what happens if we combine *this* with one of *these?*" We leave a great many such questions unanswered, pending further exploration.

Acknowledgments

Dan Piponi provided me with hours of fascinating reading, and introduced me to the notion of a free M-set monad transformer. Cale Gibbard was the first to notice the decomposition of Erwig and Kollmansberger's discrete distribution monad into $WriterT\ Prob\ []$. Nicholas Sinnott-Armstrong carefully read the first draft of this paper for clarity, and found many places that needed further explanation. Thanks also to Don Stuart, Derek Elkins, David House and other members of the Haskell community who provided me with advice and suggested new ideas. Any remaining mistakes are, of course, my own.

References

[1] "NewMonads" proposal on Haskell wiki. Retrieved on 9 June 2007 from `http://www.haskell.org/haskellwiki/NewMonads`.

[2] Rapid diagnostic testing for influenza: Information for clinical laboratory directors. Retrieved on 14 June 2007 from `http://www.cdc.gov/flu/professionals/diagnosis/rapidlab.htm`, 2006.

[3] Lloyd Allison. Types and classes of machine learning and data mining. In Michael Oudshoorn, editor, *Twenty-Sixth Australasian Computer Science Conference (ACSC2003)*, volume 16 of *Conferences in Research and Practice in Information Technology*, pages 207–215, Adelaide, Australia, February 2003.

[4] Lloyd Allison. A programming paradigm for machine learning, with a case study of bayesian networks. In *Twenty-Ninth Australasian Computer Science Conference (ACSC2006)*, volume 48 of *Conferences in Research and Practice in Information Technology*, pages 103–111, 2006.

[5] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. *The Philosophical Transactions of the Royal Society*, 53:370–418, 1763.

[6] Arnaud Doucet, Simon Godsill, and Christophe Andrieu. On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and Computing*, 10:197–208, 2000.

[7] David M. Eddy. Probabilistic reasoning in clinical medicine: Problems and opportunities. In Daniel Kahneman, Paul Slovic, and Amos Tversky, editors, *Judgment Under Uncertainty: Heuristics and Biases*, pages 249–267. Cambridge University Press, Cambridge, England, 1982.

[8] Martin Erwig and Steve Kollmansberger. Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16(1):21–34, 2006.

[9] David Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.

[10] Dieter Fox, Jeffrey Hightower, Lin Liao, Dirk Schulz, and Gaetano Borriello. Bayesian filtering for location estimation. *IEEE Pervasive Computing*, pages 24–33, September 2003.

[11] Michele Giry. A categorical approach to probability theory. In Banaschewski and Bernhard, editors, *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes in Mathematics*, pages 65–85. Springer-Verlag, 1981.

[12] Martin Grabmüller. Monad transformers step by step. Draft paper, October 2006.

[13] William Harrison and Samuel Kamin. Compilation as metacomputation: Binding time separation in modular compilers. In *5th Mathematics of Program Construction Conference (MPC2000)*, Ponte de Lima, Portugal, June 2000.

[14] C. Jones and G. D. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science (LICS)*, pages 186–195, Pacific Grove, California, June 1989.

[15] Mark Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer, 1995.

[16] Mati Kilp, Ulrich Knauer, and Alexander V. Mikhalev. *Monoids, Acts and Categories: with applications to wreath products and graphs*. Number 29 in De Gruyter Expositions in Mathematics. De Gruyter, 2000.

[17] David J. King and Philip Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Workshops in Computing, Proceedings*, Glasgow, 1992. Springer Verlag.

[18] F. William Lawvere. The category of probabilistic mappings. Unpublished, 1962.

[19] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*, pages 333–343, New York, 1995. ACM Press.

[20] Michael Mitzenmacher and Eli Upfal. *Probability and Computing*. Cambridge University Press, New York, NY, 2005.

[21] Eugenio Moggi. An abstract view of programming languages. Tech Report ECS-LFCS-90-113, Edinburgh University, 1989.

[22] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symposium on Logic in Computer Science*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[23] Peter Mueser and Donald Granberg. The Monty Hall dilemma revisited: Understanding the interaction of problem definition and decision making. Working Paper 99-06, University of Missouri, 1999.

[24] Stephen Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, page 29. Department of Computer Science, Katholieke Universiteit Leuven, 1995.

[25] Jeff Newbern. All about monads. Retrieved on 15 June 2007 from `http://www.haskell.org/all_about_monads/html/`.

[26] Raymond T. Ng and V. S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[27] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. In *32nd Annual Symposium on Principals of Programming Languages (POPL'05)*, pages 171–182, Long Beach, California, January 2005.

[28] Simon Peyton Jones et al. Haskell 98 language and libraries: The revised report, 2002.

[29] Avi Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic programming language. Computer Science Technical Report TR-12-05, Harvard Univesity, 2005.

[30] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Symposium on Principles of Programming Languages (POPL)*, pages 154–165, Portland, Oregon, January 2002.

[31] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 2003.

[32] Chung-chieh Shan. Monads for natural language semantics. In Kristina Striegnitz, editor, *Proceedings of the 2001 European Summer School in Logic, Language and Information*, pages 285–298, 2001.

[33] Chung-chieh Shan. Monad transformers. Retrieved on 14 July 2007 from `http://conway.rutgers.edu/~ccshan/wiki/blog/posts/Monad_transformers/`, 2007.

[34] Guy L. Steele, Jr. Building interpreters by composing monads. In *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, 1994. ACM Press.

[35] Marilyn vos Savant. Ask Marilyn. *Parade Magazine*, 15, 17 February 1990.

[36] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.

[37] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.