Reprinted from the Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages (POPL'02)

Stochastic Lambda Calculus and Monads of Probability Distributions

Norman Ramsey Avi Pfeffer
Division of Engineering and Applied Sciences
Harvard University

Abstract

Probability distributions are useful for expressing the meanings of probabilistic languages, which support formal modeling of and reasoning about uncertainty. Probability distributions form a monad, and the monadic definition leads to a simple, natural semantics for a stochastic lambda calculus, as well as simple, clean implementations of common queries. But the monadic implementation of the *expectation* query can be much less efficient than current best practices in probabilistic modeling. We therefore present a language of *measure terms*, which can not only denote discrete probability distributions but can also support the best known modeling techniques. We give a translation of stochastic lambda calculus into measure terms. Whether one translates into the probability monad or into measure terms, the results of the translations denote the same probability distribution.

1. Introduction

Researchers have long modeled the behavior of agents using logic, but logical models are a poor choice for dealing with the uncertainty of the real world. For dealing with inherent uncertainty and with incomplete knowledge, *probabilistic* models are better.

There are a variety of representations and reasoning techniques for probabilistic models. These techniques, which include Bayesian networks (Pearl 1988) and other kinds of graphical models (Jordan 1998), are centered around the structuring and decomposition of probability distributions. Recent work focuses on scaling up the techniques to deal with large, complex domains. Domains that require large-scale modeling techniques include medical diagnosis (Jaakkola and Jordan 1999) and military intelligence (Mahoney and Laskey 1998).

As models grow large, it becomes more important to be able to build them easily and to reuse the parts—but considered as programming languages, the techniques used to build probabilistic models are weak. We would like to draw on the large body of knowledge about programming languages to design a good language that supports probabilistic modeling. *Stochastic lambda calculus*, in which the denotations of expressions are probability distributions, not values, is a suitable basis for such a language. To express the semantics of a stochastic lambda calculus, we exploit the monadic structure of probability distributions (Giry 1981; Jones and Plotkin 1989).

The contributions of this paper are:

- We show that the probability monad leads to simple, elegant implementations of three queries commonly posed of probabilistic models: expectation, sampling, and support. Using the monad as an intermediate form simplifies proofs of desirable properties, e.g., that the sampling function draws values from any model using appropriate probabilities.
- We show that the monadic implementation of expectation is potentially much less efficient than techniques currently used in probabilistic reasoning. The problem arises because a monad does not exploit intensional properties of functions; it only applies functions. To support an alternative implementation of expectation, we translate stochastic lambda calculus into a simple language we call *measure terms*. By algebraic manipulation of measure terms, we can express *variable elimination*, which is the standard technique for efficiently computing expectation. Measure terms denote measures, and our translation into measure terms is consistent with our monadic definition of stochastic lambda calculus.

Our work has implications for both design and implementation of probabilistic languages. For design, we show that one can support efficient probabilistic reasoning simply by adding a `choose` operator to an ordinary functional language; it is not necessary to include language features that expose common implementation techniques such as variable elimination. For implementation, we show that standard techniques of programming-language implementation—monadic interpreters and algebraic manipulation of programs (including common-subexpression elimination)—can apply to probabilistic languages. Probabilistic reasoners can enjoy the benefits of higher-order, typed languages, without requiring undue effort from implementors.

POPL '02, Jan. 16-18, 2002 Portland, OR USA
©2002 ACM ISBN 1-58113-450-9/02/01...$5.00

2. Probabilistic models and queries

The simplest language we have found to describe probabilistic models is a lambda calculus in which expressions denote probability distributions. The primary means of creating interesting probability distributions is a new construct that makes a probabilistic choice. **choose** p e_1 e_2 represents a linear combination of two distributions. Operationally, to take a value from the combined distribution, with probability p we take a value from e_1 and with probability $1-p$ we take a value from e_2.

2.1. An example model

To illustrate our ideas, we present a simple model of traffic lights and drivers, using a Haskell-like notation. Traffic lights are probabilistically red, yellow, or green.

⟨*traffic example*⟩≡
```
light1 =
  dist [ 0.45 : Red, 0.1 : Yellow, 0.45 : Green ]
```

`dist` is a version of **choose** that is extended to combine two or more weighted distributions. Here it means that `light1` has value `Red` with probability 0.45, value `Yellow` with probability 0.1, and value `Green` with probability 0.45.

Drivers behave differently depending on the colors of the lights they see. A cautious driver is more likely to brake than an aggressive driver.

⟨*traffic example*⟩+≡
```
cautious_driver light =
  case light of
    Red    -> dist [ 0.2 : Braking, 0.8 : Stopped ]
    Yellow -> dist [ 0.9 : Braking, 0.1 : Driving ]
    Green  -> Driving
aggressive_driver light =
  case light of
    Red    -> dist [ 0.3 : Braking, 0.6 : Stopped,
                     0.1 : Driving ]
    Yellow -> dist [ 0.1 : Braking, 0.9 : Driving ]
    Green  -> Driving
```

We estimate that if two drivers go through a single light from different streets, there is a 90% probability of a crash.

⟨*traffic example*⟩+≡
```
crash d1 d2 light =
  [ 0.90 : d1 light         == Driving &&
           d2 (other light) == Driving,
    0.10 : False ]
  where other Red    = Green
        other Green  = Red
        other Yellow = Yellow
```

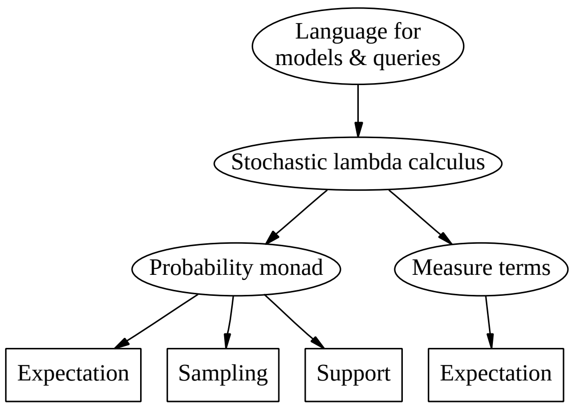
2.2. Queries

Having defined a probabilistic model, we might wish to ask questions about it. For example, if two drivers, one cautious and one aggressive, are approaching `light1`, what is the probability of a crash? This question and many others can be answered using three kinds of queries: *expectation*, *sampling*, and *support*.

The *expectation* of a function h is the mean of h over the distribution. Expectation subsumes some other queries as special cases. The mean value of a distribution is the expectation of the identity function. The probability of an outcome satisfying predicate `p` is the expectation of the function `\x -> if p x then 1 else 0`. Conditional probability can be computed from probability, since $P(p \mid q) = P(p \wedge q) \div P(q)$. In the example above, we can answer the question about the probability of a crash by building the probability distribution of `crash cautious_driver aggressive_driver light1` and computing the probability of the identity predicate.

Sampling means drawing a value from the distribution. By sampling repeatedly, we can not only approximate expectation but also get an idea of the shape of a distribution, or of some function over the distribution. Like true experimental data, samples can be fit to analytic solutions to equations; "Monte Carlo" techniques used in the physical sciences rely on sampling.

Support tells us from what subset of the entire sample space a sample might be drawn with nonzero probability. It is seldom interesting by itself, but a good implementation of support can make it easier to compute expectations efficiently. Support therefore plays a significant role in our implementation.



Ovals are representations; boxes are queries

Figure 1: Implementation paths

2.3. Implementing probabilistic models

This paper presents two representations that are useful for answering queries about probabilistic models: *probability monads* and *measure terms*. Figure 1 shows the translations involved.

1. A user writes a model and a query using a domain-specific, probabilistic language. In this paper, we take the language to be the stochastic lambda calculus that is defined formally in Section 4. In practice, we would prefer a richer language, e.g., one providing types, modules, and either an explicit fixed-point operator or recursion equations. Even in practice, however, stochastic lambda calculus is a useful intermediate form.
2. We translate the model into a more restricted target form: a value in the probability monad, or a measure term. Section 4 gives a translation into the probability monad, the semantics of which we explain in Section 3. Section 6 gives a translation into measure terms.

3. We use the target form to answer the query. The probability monad can answer all three kinds of query; measure terms are designed to answer expectation queries efficiently.

The probability monad is easy to implement in Haskell (Section 5), so we could also use Haskell as an embedded domain-specific language for probabilistic modeling. Efficient implementation of measure terms is more difficult; our current implementation is written in Objective Caml in order to exploit mutable state (Pfeffer 2001).

3. Semantics of probability and the probability monad

3.1. Measure theory

Both discrete and continuous probability are easily described using measure theory; we borrow notation from Rudin (1974). The values over which a probability distribution is defined are drawn from some space X. "Observable events" in an experiment are typically represented by subsets of X. We call these subsets the *measurable sets* of X, and we require that the entire space be measurable and that the measurable sets be closed under complement and countable union, i.e., that the measurable sets form a σ-algebra. The classical definition of a *measurable function* is a function from X to a topological space (e.g., $\mathbb{R}$) such that the inverse images of open sets are measurable. We restrict our attention to functions between measure spaces and define the measurable functions as those such that inverse images of measurable sets are measurable; that way the composition of measurable functions is measurable. Finally, a *measure* is a function μ that maps each measurable set to a real number in the range $[0, \infty]$ and that is *countably additive.* That is, if $\{A_i\}$ is a *disjoint* countable collection of measurable sets, $\mu(\bigcup_i A_i) = \sum_i \mu(A_i)$. A *probability distribution* is a measure such that $\mu(X) = 1$. We use Greek letters ν and μ to stand for measures.

Abstract integration plays a significant role in probability. If f is a real-valued, measurable function on a measurable space X with measure μ, and if A is a measurable subset of X, we write $\int_A f\,d\mu$ for the Lebesgue integral of f over set A. If e is an expression in which x appears free, we often write $\int_A e\,d\mu(x)$ instead of $\int_A (\lambda x.e)\,d\mu$.

3.2. Queries

We define our queries in terms of measure theory and abstract integration. The simplest query to define is expectation. If we have a probability distribution ν over space X, then the expectation of a function h is $\int_X h\,d\nu$.

A support of a distribution is a measurable set outside which the distribution is zero. A set S is a *support* of a distribution ν if for any measurable set A, $\nu(A) = \nu(A \cap S)$. In the language of real analysis, S is a support of ν iff ν is "concentrated on S." A good support can make it easier to compute expectations, by exploiting $\int_X h\,d\nu = \int_S h\,d\nu$.

There are many ways to formulate sampling. We could define a sampling as a function from a probability distribution to an infinite sequence of values, but we prefer to define sampling directly in terms of measure theory. Accordingly, we say that a measurable function $s : [0,1] \to X$ is a *sampling function for* ν if for any measurable set $A \subseteq X$, $\nu(A) = \mu(s^{-1}(A))$, where μ is the Lebesgue measure, which describes uniform distributions. We use this formulation for its practical value: if you give us a way of sampling uniformly on the unit interval, by applying s we'll give you a way of sampling from the probability distribution ν. The equation implies that if r is drawn uniformly from the unit interval, the probability that $s(r) \in A$ is the same as the probability that $r \in s^{-1}(A)$.

3.3. The probability monad

It has long been known that probability distributions form a monad (Lawvere 1962; Giry 1981; Jones and Plotkin 1989). In this section, we recapitulate some previous work, discussing the relationship of the monad operations to probability. The monadic bind operation, in particular, provides much of the functionality one needs to create interesting probability distributions for use in models. In Section 5, we extend the probability monad with additional operations that support queries. Throughout the paper, rather than use category-theoretic notation, we write `return` for the unit operation and $\gg\!=$ for bind (monadic extension). This notation will be more familiar to Haskell programmers and to readers of Wadler (1992).

The denotation of the unit operation is simplicity itself: `return x` stands for a distribution—that is, a measure—which assigns unit probability to any measurable set containing `x` and zero probability to any measurable set not containing `x`:

$$\mathcal{M}[\![\texttt{return x}]\!](A) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{if } x \notin A. \end{cases}$$

This measure is sometimes called the "unit mass concentrated at x." It enjoys the property that for any measurable function f,

$$\int_X f\,d\mathcal{M}[\![\texttt{return x}]\!] = f(\texttt{x}). \tag{1}$$

The proof appeals directly to the definition of Lebesgue integration. Property 1 plays a key role in the proofs of the monad laws for probability measures.

To motivate the definition of monadic bind, we appeal to conditional probability. If d denotes a probability measure over space X and k denotes a function from values in X to probability measures over space Y, then k may be interpreted as defining a "conditional probability of Y given X." Because d denotes the probability of X, applying the *chain rule* says that the *joint probability* of X and Y is equal to the probability of X times the conditional probability of Y given X. We can get the probability of Y by integrating over X. Since we wish $d \gg\!= k$ to denote the probability of Y, we define

$$\mathcal{M}[\![d \gg\!= k]\!](A) = \int_X \mathcal{M}[\![k(x)]\!](A)\,d\mathcal{M}[\![d]\!](x).$$

The notation may be confusing; the integral uses the measure $\mathcal{M}[\![d]\!]$, and the function being integrated over is $\lambda x.\mathcal{M}[\![k(x)]\!](A)$.

The following property of monadic bind plays a significant role in several of our proofs:

$$\begin{aligned} &\int_Y g(y)\,d\mathcal{M}[\![d \gg\!= k]\!](y) = \\ &\qquad \int_X \int_Y g(y)\,d\mathcal{M}[\![k(x)]\!](y)\,d\mathcal{M}[\![d]\!](x) \end{aligned} \tag{2}$$

Given properties 1 and 2, it is straightforward to prove that the definitions of `return` and $\gg=$ satisfy the monad laws.

To create some interesting distributions, we need at least one more operation in our monad. The *choose* function is a two-argument version of the `dist` used in Section 2.1. Probabilistic choice is linear combination of measures:

$$\mathcal{M}[\![choose\, p\, d_1\, d_2]\!](A) = p \cdot \mathcal{M}[\![d_1]\!](A) + (1-p) \cdot \mathcal{M}[\![d_2]\!](A),$$

where $0 \leq p \leq 1$ and the $\cdot$ symbol stands for multiplication. *choose* suffices to create any distribution with finite support. Creating other kinds of distributions, including interesting continuous distributions, would require additional operations. (One can also create continuous distributions by taking limits of discrete distributions with finite support.)

It is easy to show by induction on the structure of the monad that any value created using `return`, $\gg=$, and *choose* denotes a probability measure; the only interesting step uses property 2 with $g(y) = 1$.

3.4. Probability mass and density functions

Although it is possible to work directly with probability measures, it is often easier to work with functions that map values to their probability or probability density.

Any nonnegative, Lebesgue integrable function f defines a measure, because if μ is a bounded measure, then the function ν defined by $\nu(A) = \int_A f\, d\mu$ is also a measure. Furthermore, for any measurable A, $\int_A g\, d\nu = \int_A g \cdot f\, d\mu$. This theorem plays a significant role in the proof of the associativity law for the probability monad.

Even better, many measures can be described by a function like f. The Radon-Nikodym theorem says that if ν is bounded and if $\mu(A) = 0$ implies that $\nu(A) = 0$ (i.e., ν is *absolutely continuous* with respect to μ), then there exists a nonnegative integrable f such that $\nu(A) = \int_A f\, d\mu$. In this case, we write the function f as $\overline{\nu}$; $\overline{\nu}$ is the *Radon-Nikodym derivative* of ν.

Most implementations of probability use functions such as $\overline{\nu}$; this use is justified when the measure ν has a Radon-Nikodym derivative with respect to an underlying measure μ. Whether probability is "discrete" or "continuous" depends on the underlying measure.

- For discrete probability, countable sets are measurable, and the appropriate measure μ is the *counting measure*. This measure is useful for countable spaces; the measure of a set is the number of elements it contains. Because the counting measure assigns measure zero only to the empty set, every discrete probability distribution has a Radon-Nikodym derivative, i.e., every such distribution can be represented as a *probability-mass* function f that assigns a nonnegative probability to each element of X and which satisfies $\int_X f\, d\mu = 1$. When using the counting measure, we may use a $\sum$ symbol instead of the integral: $\int_A f\, d\mu = \sum_{x_i \in A} f(x_i)$.
- For continuous probability over real variables, which may model such processes as queueing or radioactive decay, the appropriate measure is the *Lebesgue measure* on real numbers. The Lebesgue measure of an interval is the length of the interval; see Rudin (1974) for an explanation of how to extend this measure to a much larger class of sets. Provided it assigns zero probability to sets of Lebesgue measure zero, a continuous probability distribution ν can be represented as an integrable function $\overline{\nu}$ that assigns a nonnegative *probability density* to each point of X and which satisfies $\int_X \overline{\nu}\, du = 1$. Many interesting continuous distributions fall into this class.
- For computing probabilities over infinite data structures, neither the counting measure nor the Lebesgue measure is necessarily useful. For example, if we consider infinite sequences of flips of a fair coin, there are uncountably many such sequences, and the probability of any particular sequence is zero, so the counting measure is useless. To know what questions we can sensibly ask about infinite data structures, we need to know what are the measurable sets. A good place to start is the smallest σ-algebra containing the the sets U_x defined by $U_x = \{y \mid x \sqsubseteq y\}$, where x is finite. Subject to some technical conditions, this is the Borel algebra of the Scott topology on a partially ordered set (Smyth 1983). We can use probability measures over this σ-algebra to answer such questions as "what is the probability that a sequence begins with three consecutive heads?" It is not clear to us when Radon-Nikodym derivatives (mass or density functions) exist for these kinds of measures.

Because it is often easier to work with probablity mass functions instead of measures, we give definitions of the monad operations in terms of these functions. These definitions apply only to discrete probability.

$$\begin{aligned}
\overline{\mathcal{M}[\![\texttt{return}\ v]\!]}(x) &= \begin{cases} 1, & \text{if } x = v \\ 0, & \text{if } x \neq v \end{cases} \\
\overline{\mathcal{M}[\![d \gg= k]\!]}(x) &= \textstyle\sum_v \overline{\mathcal{M}[\![d]\!]}(v) \cdot \overline{\mathcal{M}[\![k(v)]\!]}(x) \\
\overline{\mathcal{M}[\![choose\, p\, d_1\, d_2]\!]}(x) &= p \cdot \overline{\mathcal{M}[\![d_1]\!]}(x) + (1-p) \cdot \overline{\mathcal{M}[\![d_2]\!]}(x)
\end{aligned}$$

4. A stochastic lambda calculus and its semantics

Here we present a formal calculus for expressing probability distributions, and we give a denotational definition using the probability monad. To simplify the presentation, we use pairs instead of general products, binary sums instead of general sums, and binary choice (**choose**) instead of the general `dist`.

$$\begin{aligned}
e ::= &\ x \mid v \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e'\ \textbf{in}\ e \\
\mid &\ \textbf{choose}\ p\ e_1\ e_2 \\
\mid &\ (e_1, e_2) \mid e.1 \mid e.2 \\
\mid &\ \textbf{L}\ e_1 \mid \textbf{R}\ e_2 \mid \textbf{case}\ e\ e_l\ e_r
\end{aligned}$$

The **case** expression may need explanation; e is an expression of sum type, which is either left ($\textbf{L}\ v$) or right ($\textbf{R}\ v$). If left, we apply e_l to the carried value; otherwise we apply e_r. Our **case** is analogous to the `either` function found in the standard Haskell Prelude.

In our semantics, the probability monad is a type constructor M, together with unit, bind, and *choose* operations. If X is a type, $M\ X$ is the type of probability measures over X. Semantically, we wish to limit our attention to measures that restrict to continuous evaluations, so that $M\ X$ is a probabilistic powerdomain (Jones and Plotkin 1989). In Figure 2, we use the probability monad to define the semantics of the stochastic lambda calculus. By using suitable domains with the translation in Figure 2, we can make a denotational semantics for the stochastic

$$\begin{array}{ll}
\mathcal{P}[\![x]\!]\rho & = \texttt{return}\ (\rho\ x) \\
\mathcal{P}[\![v]\!]\rho & = \texttt{return}\ v \\
\mathcal{P}[\![\lambda x.e]\!]\rho & = \texttt{return}\ (\lambda v.\mathcal{P}[\![e]\!]\rho\{x \mapsto v\}) \\
\mathcal{P}[\![\textbf{let}\ x = e'\ \textbf{in}\ e]\!]\rho & = \mathcal{P}[\![e']\!]\rho \ggg= \lambda v.\mathcal{P}[\![e]\!]\rho\{x \mapsto v\} \\
\mathcal{P}[\![e_1\ e_2]\!]\rho & = \mathcal{P}[\![e_1]\!]\rho \ggg= \lambda v_1.\mathcal{P}[\![e_2]\!]\rho \ggg= \lambda v_2.v_1 v_2 \\
\mathcal{P}[\![(e_1, e_2)]\!]\rho & = \mathcal{P}[\![e_1]\!]\rho \ggg= \lambda v_1.\mathcal{P}[\![e_2]\!]\rho \ggg= \lambda v_2.\texttt{return}\ (v_1, v_2) \\
\mathcal{P}[\![e.1]\!]\rho & = \mathcal{P}[\![e]\!]\rho \ggg= (\texttt{return} \circ \texttt{fst}) \\
\mathcal{P}[\![e.2]\!]\rho & = \mathcal{P}[\![e]\!]\rho \ggg= (\texttt{return} \circ \texttt{snd}) \\
\mathcal{P}[\![\textbf{choose}\ p\ e_1\ e_2]\!]\rho & = \mathit{choose}\ p\ (\mathcal{P}[\![e_1]\!]\rho)\ (\mathcal{P}[\![e_2]\!]\rho) \\
\mathcal{P}[\![\textbf{L}\ e]\!]\rho & = \mathcal{P}[\![e]\!]\rho \ggg= (\texttt{return} \circ \texttt{Left}) \\
\mathcal{P}[\![\textbf{R}\ e]\!]\rho & = \mathcal{P}[\![e]\!]\rho \ggg= (\texttt{return} \circ \texttt{Right}) \\
\mathcal{P}[\![\textbf{case}\ e\ e_l\ e_r]\!]\rho & = \mathcal{P}[\![e]\!]\rho \ggg= \texttt{either}\ (\lambda v.\mathcal{P}[\![e_l]\!]\rho \ggg= \lambda f.f\ v)\ (\lambda v.\mathcal{P}[\![e_r]\!]\rho \ggg= \lambda f.f\ v)
\end{array}$$

Figure 2: Translation of stochastic calculus into the probability monad

lambda calculus (Jones 1990, Chapter 8). This translation has two significant properties:

- The denotations of expressions are not the same kinds of objects that appear in environments. Expressions denote probability distributions, but environments bind identifiers to *values*, not to distributions.
- The denotation of a lambda abstraction of type `a -> b` is a function from values of type `a` to *probability distributions* over values of type `b`.

These properties, as well as the rest of the translation, are what you would expect to find in a monadic semantics of a call-by-value language with imperative features, except that we use the probability monad instead of a state or I/O monad. We expect that an analogous lazy semantics should also exist, but we have not explored that avenue.

Denotational definitions are out of fashion. Why don't we have a syntactic theory with an operational semantics? Because a denotational definition makes it easier to show how the mathematical foundations of probability apply to the semantics of probabilistic languages. Our denotational definitions also correspond closely to the structures of our implementations. It is a drawback that a reader has to work harder to see that the denotation of $(\lambda x.x - x)(\textbf{choose}\ \frac{1}{2}\ 0\ 1)$ is the unit mass concentrated at 0. In other words, when reducing lambda terms, it is not permissible to duplicate a redex.

5. Probability monads in Haskell

Using Haskell, we have implemented the probability monad for discrete distributions. We have taken advantage of Haskell's system of type and constructor classes to provide three specialized implementations: one for each kind of query.

The probability monad provides `choose`; it inherits `return` and `>>=` from class `Monad`. For simplicity, we represent probabilities as double-precision floating-point numbers. By using Haskell's type classes, we could support more general representations, but the details would be distracting.

⟨*probability monads*⟩≡
```
type Probability = Double    -- number from 0 to 1
class Monad m => ProbabilityMonad m where
    choose :: Probability -> m a -> m a -> m a
```

Like any other monad, the probability monad needs at least one operation that observes what is inside a monadic computation (Hughes 1995, §5). The obvious observations are our three queries. Rather than require one implementation to support all three queries, however, we structure the code so that we can provide three different implementations, one for each kind of query.

Since distributions created with this interface have countable support, it is easiest to represent a support of a distribution by a set of elements. To make the code more readable, we use lists instead of sets, and we permit that some elements appear more than once; at need, Haskell programmers can eliminate duplicates using `List.nub`.

⟨*probability monads*⟩+≡
```
class ProbabilityMonad m => SupportMonad m where
  support :: m a -> [a]
  -- support (return x) = [x]
  -- support (d >>= k) =
  --   concat [support (k x) | x <- support d]
  -- support (choose p d d') =
  --   support d ++ support d'
```

The comments show algebraic laws that `support` must satisfy; the support monad is a simple extension of the list monad. In the law we give for `choose`, the value `support (choose p d d')` does not depend on `p`, even when `p` is 0 or 1. Indeed, if we remove `p` from the signature of `choose`, we get the well-known nondeterminism monad.

Our law for `choose` is sound, and we have chosen it for its simplicity, but it is really an oversimplification. In practice it is important to add the side condition $0 < \texttt{p} < 1$ and to add laws to cover the cases $\texttt{p} = 0$ and $\texttt{p} = 1$. Probabilities of 0 and 1 can arise in real models from, e.g., idiomatic translations of Bayesian networks, and cutting down the support as much as possible can save significant computation.

It is easy to show that any implementation of `support` satisfying the laws above produces a list of values that, when taken as a set, forms a support for the measure denoted by the monadic value. The proof relies on the discreteness of the measure; the inductive hypothesis is that $\mathcal{M}[\![d]\!](x) > 0$ implies $x \in \texttt{support}\ d$.

For simplicity, we define expectation only of real-valued functions. It is not difficult to define a version of the expectation monad that can compute the expectation of any function whose range is a vector space over the reals.

The expectation monad and its laws are as follows.

```
⟨probability monads⟩+≡
class ProbabilityMonad m => ExpMonad m where
  expectation :: (a -> Double) -> m a -> Double
  -- expectation h (return x) = h x
  -- expectation h (d >>= k)  = expectation g d
  --           where g x = expectation h (k x)
  -- expectation h (choose p d d') =
  --       p * expectation h d +
  --         (1-p) * expectation h d'
```

Using property 2 from Section 3.3, it is easy to prove that any implementation of `expectation` satisfying these laws computes expectation as defined by the measure.

Stipulating true real-number arithmetic, with infinitely many bits of precision, we present laws for a sampling function. If d is a distribution, $\texttt{fst} \circ \texttt{sample}\ d$ is a sampling function in the sense of Section 3.2.

```
⟨probability monads⟩+≡
  -- sample (return x) r = (x, r)
  -- sample (d >>= k)  r =
  --   let (x, r') = sample d r in sample (k x) r'
  -- sample (choose p d d') r =
  --   if r < p then sample d (r/p)
  --   else sample d' ((1-r)/(1-p))
```

The law for `choose` is the interesting law; it uses as many bits of precision as are needed to compare `r` and `p`, then renormalizes so that the remaining bits can be used for further samples. The computation is like what happens to the output of an arithmetic coder (Witten, Neal, and Cleary 1987).

The proof of correctness of the sampling laws is the most difficult proof in this paper. The key is finding a good induction hypothesis: for any distribution d and any real function f defined on the product space $X \times \mathbb{R}$,

$$\int_I f(\texttt{sample}\ d\ r)\, d\mu(r) = \int_I \int_X f(x, r)\, d\mathcal{M}[\![d]\!](x)\, d\mu(r),$$

where I is the unit interval and μ is Lebesgue measure. From this hypothesis it is straightforward to show that $\mu((\texttt{fst}\circ\texttt{sample}\ d)^{-1}(A)) = \mathcal{M}[\![d]\!](A)$, so that $\texttt{fst}\circ\texttt{sample}\ d$ is a sampling function for $\mathcal{M}[\![d]\!]$.

This definition of sampling would not be very useful in an implementation, because it is based on a single random number with arbitrarily many bits of precision. It is more consistent with Haskell's standard library to use a random-number *generator* for sampling:

```
⟨probability monads⟩+≡
class ProbabilityMonad m => SamplingMonad m where
  sample :: RandomGen g => m a -> g -> (a, g)
  -- sample (return x) g = (x, g)
  -- sample (d >>= k)  g =
  --   let (x, g') = sample d g in sample (k x) g'
  -- sample (choose p d d') g =
  --   let (x, g') = random g in
  --     sample (if x < p then d else d') g'
```

Although we can give a denotational semantics to the sampling monad purely in terms of sampling functions, the sampling monad has an appealing operational interpretation as a monad of *random experiments.* If m is in the class `SamplingMonad`, then a value of type $m\ \alpha$ represents an experiment that returns a value of type α. The unit computation `return` x represents the experiment that always returns x. A computation produced by bind, $d \gg\!= k$, represents the experiment that begins by running d to generate an intermediate value, applies k to the value to get a second experiment, and returns the result of the second experiment. *choose* $p\, d_1\, d_2$ represents the experiment that runs d_1 with probability p and d_2 with probability $1 - p$.

The algebraic laws above lead directly to implementations of the monads. Additional notation is needed to make legal Haskell; the code appears in Appendix A.

5.1. Performance

The monads above compute support and sampling about as efficiently as possible, asymptotically speaking. The expectation monad, by contrast, is inherently inefficient, and for some terms in the stochastic lambda calculus, it may perform exponentially worse than other algorithms. The problem with the monadic computation of expectation is that when we compute `expectation` $h\ d$, we don't have any information about the structure of the function h. We must therefore apply h to every value in its domain,[1] at a cost proportional to the size of that domain. But if h is defined over a product domain, it may be possible to compute the expectation of h at a lower cost, depending on the structure of h. For example, if the domain of h is the product space $X \times Y$, and if there exist functions $h_{1,i}$ and $h_{2,i}$ such that $h(x, y) = \sum_i h_{1,i}(x)h_{2,i}(y)$, and if μ can be similarly split, then $\int_{X\times Y} h\, d\mu(x, y) = \sum_i (\int_X h_{1,i}(x)\, d\mu_1(x)) \cdot (\int_Y h_{2,i}(y)\, d\mu_2(y))$. The cost of computing the left-hand side is $O(|X \times Y|)$, but the cost of computing the right-hand side is $O(|X| + |Y|)$. If we are computing the expectation of a function of a large number of variables (e.g., the expected number of heads in a sequence of 10 coin flips), the monad can take exponential time to solve a linear problem.

Many functions over which we might wish to take expectations have a structure that can be exploited. For example, in a probabilistic grammar for natural language, a model might say that a sentence is made up of a noun phrase and a verb phrase. In many models, such as probabilistic context-free grammars (Charniak 1993), the two phrases define *independent* probability distributions, so the probability distribution over the string generated by the verb phrase does not depend on that generated by the noun phrase. If we want to compute the expected number of words in a sentence, h is the number of words in the noun phrase plus the number of words in the verb phrase, and it has the structure required. Even if the noun phrase and verb phrase are not quite independent, but the verb phrase is influenced by the first word of the noun phrase, the verb phrase is still conditionally independent of the remaining words of the noun phrase given the first word, and the independence can be exploited. The probability monad cannot exploit the independence, because it must produce the entire sentence, including both phrases, before it can apply h.

[1]Actually it suffices to apply h only to those values that are in the support of d. This refinement is important in practice, but it does not affect the asymptotic costs.

$$t ::= t_1 \times t_2 \mid t_1 + t_2 \mid \textstyle\sum_{y::Y} t \mid \langle \phi : w \rangle$$

$$\begin{array}{ll}
\overline{\mathcal{T}[\![t_1 \times t_2]\!]\Gamma}(\rho) & = \overline{\mathcal{T}[\![t_1]\!]\Gamma}(\rho) \cdot \overline{\mathcal{T}[\![t_2]\!]\Gamma}(\rho) \\
\overline{\mathcal{T}[\![t_1 + t_2]\!]\Gamma}(\rho) & = \overline{\mathcal{T}[\![t_1]\!]\Gamma}(\rho) + \overline{\mathcal{T}[\![t_2]\!]\Gamma}(\rho) \\
\overline{\mathcal{T}[\![\sum_{y::Y} t]\!]\Gamma}(\rho) & = \sum_{v \in Y} \overline{\mathcal{T}[\![t]\!](\{y :: Y\} \uplus \Gamma)}(\rho\{y \mapsto v\}) \\
\overline{\mathcal{T}[\![\langle \phi : w \rangle]\!]\Gamma}(\rho) & = w \cdot sat(\phi, \rho) \\
sat(\phi, \rho) & = \begin{cases} 1, \text{ if formula } \phi \text{ is satisfied by the assignment } \rho \\ 0, \text{ otherwise} \end{cases}
\end{array}$$

The bound variable in a sum must be distinct from other variables; the union $\{y :: Y\} \uplus \Gamma$ is defined only if variable y does not appear in any pair in Γ.

Figure 3: Syntax and semantics of measure terms

6. A term language for computing discrete expectation

The probability monad leads to elegant implementations of our three queries, but a monadic computation of expectation over a product space can take time exponential in the number of dimensions in the space. The rest of this paper is devoted to a different abstraction: *measure terms*. Using measure terms, we can expose the structure of h and rewrite sums (integrals) over product spaces into products of sums, thereby reducing the cost of computing the expectation of h.

Measure terms define measures over product spaces. The simplest kind of term is a *weighted formula* $\langle \phi : w \rangle$, which is a generalization of unit mass; it assigns a real, nonnegative weight w to a set in the product space. The set is defined by a formula ϕ, which is simply a predicate. The angle brackets and colon serve only to delimit the weighted formula; they have no semantic significance. Measure terms may be multiplied and added, and we can also sum over variables. The syntax of measure terms is as follows:

$$t ::= \langle \phi : w \rangle \mid t_1 \times t_2 \mid t_1 + t_2 \mid \textstyle\sum_{x::X} t$$

We write terms assuming that $\times$ binds more tightly than $+$, and the scope of $\sum$ extends as far to the right as possible. Our notation has one subtlety; the colons in $\sum_{x::X}$ indicate that this summation symbol is syntax. When we mean summation in a semantic domain, over a measurable space X, we write $\sum_{x \in X}$.

One reason to use measure terms is be able to choose the order in which we do sums. We therefore use a representation of product spaces in which we identify dimensions by name, not by the way they are ordered. In particular, we represent an n-dimensional product space as a set of pairs $x_i :: X_i$, where x_i is a name and X_i is a measurable space. We represent a value in this product space by an *environment* mapping names to values such that the domain of the environment is $\{x_i \mid 1 \le i \le n\}$. Every name defined in the environment identifies one dimension of the product space.

A measure term denotes a measure over a product space, but its denotation may depend on context, i.e., what product space we are interested in. We therefore specify a product space when giving the semantics of a term. By analogy with type environments, we write this product space as $\Gamma = \{x_i :: X_i \mid 1 \le i \le n\}$. We write a value in the product space as $\rho = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$. Figure 3 gives the meanings of terms. The function $\overline{\mathcal{T}[\![t]\!]\Gamma}$ defines a probability-mass function on Γ; to get a measure, we integrate using the counting measure. That is, $\mathcal{T}[\![t]\!]\Gamma(A) = \sum_{\rho \in A} \overline{\mathcal{T}[\![t]\!]\Gamma}(\rho)$. The context Γ plays no role in the evaluation of $\mathcal{T}$, but we need it to prove that our translation of stochastic lambda calculus into measure terms is equivalent to our translation into the probability monad.

Measure terms obey useful algebraic laws:

- The term $\langle \textbf{true} : 1 \rangle$ is a unit of term product $\times$, and $\langle \textbf{false} : w \rangle$ is a unit of term addition $+$.
- Term product $\times$ and sum $+$ obey associative, commutative, and distributive laws.
- We can rename variables in sums; the x in $\sum_{x::X}$ is a binding instance.
- We can interchange sums over different variables.
- We can sometimes move terms outside sums, i.e., we have $\sum_{x::X} t_1 \times t_2 = t_1 \times \sum_{x::X} t_2$, provided x is not free in t_1.

The soundness of these laws follows immediately from the definition of $\mathcal{T}$.

The laws make it possible to implement *variable elimination*, which is a code-improving transformation that reduces work by moving products outside sums. A full treatment of variable elimination is beyond the scope of this paper, but we can state the idea briefly. If each of two terms t_1 and t_2 contains free variables that do not appear free in the other, the laws enable us to convert a sum of products to a product of sums. For example, if x_1 is not free in t_2 and x_2 is not free in t_1, then $\sum_{x_1::X_1} \sum_{x_2::X_2} t_1 \times t_2 = (\sum_{x_1::X_1} t_1) \times (\sum_{x_2::X_2} t_2)$. The cost of computing $\mathcal{T}$ on the left-hand side is $O(|X_1| \cdot |X_2|)$, but the cost of computing $\mathcal{T}$ on the right-hand side is $O(|X_1| + |X_2|)$. Rewriting a term to minimize the cost of computing $\mathcal{T}$ is an NP-hard problem, but there are heuristics that work well in practice (Jensen 1996).

6.1. Measure terms for probability

To represent a probability distribution, we use a measure term with a distinguished free variable, arbitrarily called $*$ (pronounced "result"). Figure 4 shows how to translate a term in the stochastic lambda calculus into a measure term with the additional free variable $*$. We have left the domains of variables out of the sums; the domains are computed using the support function $\mathcal{S}$. Figure 4 uses two metanotations that may require explanation. We write substitution using superscripts and subscripts; t_x^e stands for

$$
\begin{aligned}
\mathcal{E}[\![x]\!]\rho &= \langle * = x : 1\rangle \\
\mathcal{E}[\![v]\!]\rho &= \langle * = v : 1\rangle \\
\mathcal{E}[\![\lambda x.e]\!]\rho &= \langle * = \lambda v.\mathcal{E}[\![e]\!]\rho\{x \mapsto \{v\}\} : 1\rangle \\
\mathcal{E}[\![\mathbf{let}\ x = e'\ \mathbf{in}\ e]\!]\rho &= \sum_x (\mathcal{E}[\![e']\!]\rho)^x_* \times \mathcal{E}[\![e]\!]\rho\{x \mapsto \mathcal{S}[\![e']\!]\rho\} \\
\mathcal{E}[\![e_1\ e_2]\!]\rho &= \sum_{x_f}\sum_{x_a} (\mathcal{E}[\![e_1]\!]\rho)^{x_f}_* \times (\mathcal{E}[\![e_2]\!]\rho)^{x_a}_* \times \\
&\qquad \prod_{v_f \in \mathcal{S}[\![e_1]\!]\rho,\, v_a \in \mathcal{S}[\![e_2]\!]\rho} \Big((v_f\ v_a \times \langle x_f = v_f \wedge x_a = v_a : 1\rangle) + \langle \neg(x_f = v_f \wedge x_a = v_a) : 1\rangle\Big) \\
\mathcal{E}[\![\mathbf{choose}\ p\ e_1\ e_2]\!]\rho &= (\mathcal{E}[\![e_1]\!]\rho \times \langle \mathbf{true} : p\rangle) + (\mathcal{E}[\![e_2]\!]\rho \times \langle \mathbf{true} : 1-p\rangle) \\
\mathcal{E}[\![(e_1, e_2)]\!]\rho &= (\mathcal{E}[\![e_1]\!]\rho)^{*.1}_* \times (\mathcal{E}[\![e_2]\!]\rho)^{*.2}_* \\
\mathcal{E}[\![e.1]\!]\rho &= \sum_{x_{\text{new}}} (\mathcal{E}[\![e]\!]\rho)^{(*, x_{\text{new}})}_* \\
\mathcal{E}[\![e.2]\!]\rho &= \sum_{x_{\text{new}}} (\mathcal{E}[\![e]\!]\rho)^{(x_{\text{new}}, *)}_* \\
\mathcal{E}[\![\mathbf{L}\ e]\!]\rho &= (\mathcal{E}[\![e]\!]\rho)^{\texttt{left}\ *}_* \times \langle \texttt{isLeft}\ * : 1\rangle \\
\mathcal{E}[\![\mathbf{R}\ e]\!]\rho &= (\mathcal{E}[\![e]\!]\rho)^{\texttt{right}\ *}_* \times \langle \texttt{isRight}\ * : 1\rangle \\
\mathcal{E}[\![\mathbf{case}\ e\ e_l\ e_r]\!]\rho &= \mathcal{E}[\![\texttt{either}\ e_l\ e_r\ e]\!]\rho
\end{aligned}
$$

Here $\mathcal{S}$ is the *support* function, which gives us the set of possible values to which an expression e could evaluate; the environment ρ is used only to compute support. The variables x_{new}, x_f, and x_a are unique, fresh variables. We notate substitution using superscripts and subscripts; t^e_x stands for the measure term t with expression e substituted for variable x. Functions `either`, `left`, `right`, `isLeft`, and `isRight` are predefined functions that support sum types; the name `either` represents a literal value, not a variable.

Figure 4: Translation of stochastic calculus into terms

the measure term t with expression e substituted for variable x. In the rule for application, the product symbol $\prod$ is a metalanguage symbol and not part of the measure-term language; it stands for a large multiplication. This multiplication is a device for applying functions to get measure terms. If we write the multiplication as $t = \prod \cdots$, then given any record ρ that maps x_f to v_f and x_a to v_a, $\mathcal{T}[\![t]\!]\Gamma(\rho) = \mathcal{T}[\![v_f\ v_a]\!]\Gamma(\rho)$.

Making variable elimination work effectively on the results of the translation requires two steps not shown in Figure 4: introducing multiple variables for product spaces and manipulating the translation of **choose** to keep independent terms separate.

Because the single free variable $*$ often stands for a value in a product space, we need a way to "split up" the variable into its constituent parts. The product spaces $\Gamma \uplus \{x :: X_1 \times X_2\}$ and $\Gamma \uplus \{x_1 :: X_1, x_2 :: X_2\}$ are isomorphic; we take them to be equivalent, and we use the equality

$$
\begin{aligned}
&\mathcal{T}[\![t]\!](\Gamma \uplus \{x :: X_1 \times X_2\})(\rho\{x \mapsto (v_1, v_2)\}) = \\
&\qquad \mathcal{T}[\![t^{(x_1,x_2)}_x]\!](\Gamma \uplus \{x_1 :: X_1, x_2 :: X_2\})(\rho\{x_1 \mapsto v_1, x_2 \mapsto v_2\})
\end{aligned}
$$

to change variables within terms. This equality supports the following additional algebraic law:

$$\sum_x t = \sum_{x_1}\sum_{x_2} t^{(x_1,x_2)}_x.$$

In the translation of **choose** shown in Figure 4, terms $\mathcal{E}[\![e_1]\!]\rho$ and $\mathcal{E}[\![e_2]\!]\rho$ are combined using addition. As a consequence, even if each of the terms contains free variables not found in the other, we will not be able to move the addition outside sums over those free variables. In our implementation, we apply the following law to the translation of **choose**:

$$
\begin{aligned}
&t \times \langle \mathbf{true} : p\rangle + t' \times \langle \mathbf{true} : 1-p\rangle = \\
&\quad \textstyle\sum_{x_{\text{new}} :: \texttt{Bool}} (t \times \langle x_{\text{new}} : p\rangle + \langle \neg x_{\text{new}} : 1\rangle) \times \\
&\qquad (t' \times \langle \neg x_{\text{new}} : 1-p\rangle + \langle x_{\text{new}} : 1\rangle).
\end{aligned}
$$

On the right-hand side, we now have a chance to move other sums inside $\sum_{x_{\text{new}}}$.

6.2. Equivalence of two translations

A term in the stochastic lambda calculus denotes the same probability measure whether we compute the measure using the probability monad or using measure terms. This claim is central to our contribution, linking up elegant techniques for defining languages with efficient techniques for probabilistic reasoning.

The proof requires some technicalities associated with denotations of functions. The problem is this: in the $\mathcal{P}$ translation, the denotation of a lambda abstraction is made using a function returning a value in the probability monad; in the $\mathcal{E}$ translation, the denotation of a lambda abstraction is made using a function returning a measure term; to prove them equivalent, we need to reason about functions returning probability measures. We therefore have three different spaces of values, which we call M (monadic) space, T (term) space, and V (value) space. To show the two translations are equivalent, we need to be able to map both M space and T space into V space.

We define the mappings we need using a type-indexed family of transformations we call *lift*. If v is an atomic, zero-order value (integer, string, etc), then *lift* $\mathcal{F}$ $v = v$. If v is a pair, then *lift* $\mathcal{F}$ $(v_1, v_2) = ($*lift* $\mathcal{F}$ v_1, *lift* $\mathcal{F}$ $v_2)$, and similarly for sum types. Finally, if v is a function, we define *lift* such that $(\mathit{lift}\ \mathcal{F}\ v)\ (\mathit{lift}\ \mathcal{F}\ v') = \mathcal{F}(v\ v')$. The *lift* transformation is closely related to the *reify* transformation used in type-directed partial evaluation (Danvy 1996).

Having defined *lift*, we extend it to expressions, formulas, and environments by saying that $\mathit{lift}\ \mathcal{F}\ e$ is the expression obtained by replacing all *literal values* in e with their lifted forms, and similarly for formulas and environments. In particular, when $\lambda x.e$ is syntax, $\mathit{lift}\ \mathcal{F}\ (\lambda x.e) = \lambda x.\mathit{lift}\ \mathcal{F}\ e$. Finally, we have to adjust the definitions of $\mathcal{M}$ and $\mathcal{T}$.

$$\begin{array}{rcl}
\overline{\mathcal{M}[\![\mathtt{return}\ v]\!]}(x) & = & \begin{cases} 1, & \text{if } x = \mathit{lift}\ \mathcal{M}\ v \\ 0, & \text{if } x \neq \mathit{lift}\ \mathcal{M}\ v \end{cases} \\
\overline{\mathcal{M}[\![d \gg\!= k]\!]}(x) & = & \sum_v \overline{\mathcal{M}[\![d]\!]}(\mathit{lift}\ \mathcal{M}\ v) \cdot \overline{\mathcal{M}[\![k(v)]\!]}(x) \\
\overline{\mathcal{T}[\![\langle \phi : w \rangle]\!]}\Gamma(\rho) & = & w \cdot \mathit{sat}(\mathit{lift}\ \mathcal{T}\ \phi, \rho)
\end{array}$$

With the new definitions, a lambda term e that is translated via the probability monad has literal values that live in M space, a lambda term that is translated via measure terms has literal values that live in T space, but the denotations of both are probability measures that apply to values in V space.

With *lift*, we have the machinery we need to state that the two translations are equivalent. The proof is by structural induction on lambda terms, and the induction hypothesis is that if e_m and e_t are two lambda terms such that $\mathit{lift}\ \mathcal{M}\ e_m = \mathit{lift}\ \mathcal{T}\ e_t$, then

$$\begin{aligned}
&\overline{\mathcal{M}(\mathcal{P}[\![e_m]\!]\rho)}(v) = \\
&\qquad \overline{\mathcal{T}(\mathcal{E}[\![e_t]\!]\rho)(\Gamma_\rho \uplus \{* :: Y\})}((\mathit{lift}\ \mathcal{M}\ \rho)\{* \mapsto v\})
\end{aligned}$$

where Γ_ρ gives the names and types of variables in the domain of ρ, and Y is the type of e (and also the measurable space from which v is drawn). It follows that if e is a lambda term in which the only literals are zero-order literals such as integers, we can translate it either way and get the same probability distribution. A lemma that applies frequently to the proof is that adding variables to the environment of a term doesn't matter:

$$\overline{\mathcal{T}[\![t]\!]\Gamma}(\rho) = \overline{\mathcal{T}[\![t]\!](\Gamma \cup \{y\})}(\rho\{y \mapsto v\})$$

provided y does not appear in Γ, in the domain of ρ, or free in t. In this lemma, v is arbitrary.

6.3. Expectation

Using measure terms, we can speed up the computation of the expectation of a function h, provided we expose the structure of h to the variable-elimination algorithm. Our implementation doesn't compute expectations of general functions expressed in lambda terms; it computes expectations only of functions that can be expressed in the form $h(v) = w_1 \cdot \mathit{sat}(\phi_1, \{* \mapsto v\}) + \cdots + w_n \cdot \mathit{sat}(\phi_n, \{* \mapsto v\})$. For such a function, we define $\mathtt{expectation}'\ h\ t = \overline{\mathcal{T}[\![\sum_* t \times (\langle \phi_1 : w_1 \rangle + \cdots + \langle \phi_n : w_n \rangle)]\!]\{\}}\{\}$. Using the equivalence of the previous section, it is not hard to show that for a suitable closed lambda-term e,

$$\mathtt{expectation}'\ h\ (\mathcal{E}[\![e]\!]\{\}) = \mathtt{expectation}\ h\ (\mathcal{P}[\![e]\!]\{\}).$$

Using variable elimination on the left-hand side can produce exponential speedups for some functions h.

7. Related work

Monads are used both in the study of language features and in the implementation of programming languages; Wadler (1992) introduces and motivates the topic. Monads come to us from category theory, and category theorists have long been aware of the monadic structure of probability (Lawvere 1962; Giry 1981). The support and sampling monads are also well known in the programming community, especially the support monad, because it describes nondeterministic choice. It appears that the expectation monad is not well known, perhaps because of its inherent inefficiency.

A tool called QuickCheck uses the sampling monad to generate random values for testing functional programs (Claessen and Hughes 2000). QuickCheck's probabilistic-choice operator, although also called `choose`, is somewhat different from ours; it produces an integer distributed uniformly over a specified range. Either version of `choose` can be simulated using the other. QuickCheck also provides a `frequency` function, which is analogous to `dist`. Claessen and Hughes (2000) also presents *generator transformers*, which use existing generators to build new ones in ways that cannot be done using monadic bind alone. It is not obvious how the idea relates to the general probability monad.

Substantial effort has been devoted to developing Scott-style domain theory that could help specify semantics of probabilistic languages. Saheb-Djahromi (1980) shows the complete-partial-order structure of probability distributions on a domain. Jones and Plotkin (1989) uses evaluations rather than measures to build probabilistic powerdomains. Jones's (1990) doctoral thesis is slightly more accessible to the amateur; Section 1.4 provides a brief guide to the literature.

It would be pleasant to extend our calculus with recursive functions, recursion equations, or a fixed-point operator, but to give a careful semantics to such an extended calculus would require domain theory, category theory, and analysis that are beyond the scope of this paper. The particularly sticky bits have to do with permitting probability distributions over functions, which we wish to do in our models. We have identified two related languages that support recursion.

Saheb-Djahromi (1978) presents a probabilistic version of LCF, in which expressions of base types (integer or Boolean) denote probability distributions but expressions of function type denote functions, not distributions over functions. The language includes both call-by-name and call-by-value abstraction constructs; only values of base types may be passed to call-by-value functions. The paper presents both denotational and operational semantics and shows them equivalent.

Jones (1990), Chapter 8 presents a call-by-value language that is much closer to ours. The major distinctions are that the language includes a fixed-point operator and that it has not one but two syntactic categories: expressions and function expressions. Expressions denote probability distributions; function expressions denote functions from values to probability distributions. The syntax is structured such that only function expressions can be applied, and the fixed-point operator can be applied only to function expressions, not to ordinary expressions. Using syntax whose meaning corresponds to the monadic unit operation, every function expression can be made into an ordinary expression, but there is no reverse path. Thus, although the language does support recursive functions, it is not possible to apply a function that is stored in a tuple or passed as an argument. The thesis presents both denotational and operational semantics for the language and shows them equivalent. To avoid difficulties with product spaces, the denotational semantics uses evaluations, not measures, to represent probability distributions. Modulo this difference and our omission of recursive functions, the denotational semantics appears to be isomorphic to ours.

Benveniste et al. (1995) and Gupta, Jagadeesan, and Panangaden (1999) discuss languages for the description of concurrent probabilistic systems. Their work differs in flavor from ours, since it combines the probabilistic specification of stochastic behavior with unquantified, constraint-based non-determinism. As a result, a program may or may not define a single probability distribution. In our language, there is only probabilistic non-determinism, and a program defines a unique probability measure.

There is a significant body of work available on variable elimination. Pearl (1988) popularized graphical models, including Bayesian networks, as well as the polytree reasoning algorithm. Commercial reasoning systems commonly use the junction-tree algorithm of Lauritzen and Spiegelhalter (1988). Zhang and Poole (1994) describes a variable-elimination algorithm for Bayesian networks; Li and d'Ambrosio (1994) presents an algebraic variation. Dechter (1996) shows that the polytree and junction-tree algorithms are also forms of variable elimination, and that the variable-elimination framework unites probabilistic reasoning with a variety of other tasks such as constraint satisfaction. Arnborg (1985) lays the graph-theoretic foundations for variable elimination.

Most implementations of variable elimination seem to be based on more specialized representations than we use. For example, the basic unit of representation may be a *table* that corresponds to a measure term of the form $\sum_{i=1}^{n} \langle \wedge_{j=1}^{m} \pi_j = v_{ij} : w_i \rangle$, where π is a path $x.k_1.k_2 \ldots k_n$ and v_{ij} is a value or "don't care." The generality of measure terms significantly simplifies the implementation, and it appears not to impose unacceptable efficiency costs.

Learning algorithms that use variable elimination often combine it with memoization (frequently called "caching" in the probabilistic-reasoning literature). We believe that we can achieve similar performance gains, without introducing mutable state into an implementation, by introducing let-binding for measure terms and by using hash-consing to implement common-subexpression elimination.

Other researchers in probabilistic modeling have used languages that resemble our stochastic lambda calculus. Koller, McAllester, and Pfeffer (1997) presents a simple Lisp-like language with a coin-toss function, giving it an operational semantics based on sampling experiments. Muggleton (2001) presents a similar extension to logic programs. Luger and Pless (2001) proposes using a stochastic lambda calculus, with a traditional reduction semantics, as a foundation for probabilistic-modeling languages. Lambda terms are reduced to sets of (value, probability) pairs, and additional rules are needed to distribute function application over these sets; in our framework, the monadic bind solves this problem. The paper argues that deBruijn indices should make it easy to identify equal values and to support memoization.

8. Discussion

We have elucidated connections between the measure-theoretic basis of probability, monadic techniques for describing programming languages, and variable-elimination techniques used for efficient probabilistic reasoning. Using a monad to describe the stochastic lambda calculus is not only entertaining—it reduces proof obligations. For example, given our theorems about the various probability monads, we can take *any* probabilistic language, translate it into the probability monad, and be sure sampling is consistent with expectation. Using a monad enables us to separate the structure of our semantic domain from the details of any particular probabilistic language.

It is not clear whether we can retain these advantages and also exploit the greater efficiency of measure terms. We can create an instance of the probability monad that produces measure terms, but the cost of the obvious algorithm is proportional to the size of the product space. The open question is whether we can create an instance of the probability monad that produces measure terms at a cost no greater than the cost of evaluating those terms *after* variable elimination. Techniques inspired by type-directed partial evaluation, which can produce abstract syntax for native functions, might solve this problem (Danvy 1996).

Another challenge is to add recursion to our calculus. From the point of view of the probability monad, this would appear to present few difficulties. Suppose we wish to define a recursive function $f = \lambda x.e$, where both f and x appear free in e. The key is that when we consider the meaning of e, f must stand for a function, not a probability distribution over functions. It would therefore be difficult to use the classical style or fixed-point combinator. It is easy, however, to use a style that takes fixed points only of syntactic lambda abstractions; we write $\mathbf{fix}\ f\ x\ e$, with $\mathcal{P}[\![\mathbf{fix}\ f\ x\ e]\!]\rho = \mathit{fix}(\lambda w.\lambda v.\mathcal{P}[\![e]\!]\rho\{f \mapsto w, x \mapsto v\}$, where $\mathit{fix}(g) = \bigsqcup_{i=0}^{\infty} g^{(i)}(\bot)$. Generalizing such a construct to mutually recursive functions is straightforward (Appel 1992). Jones (1990) explains why the fixed point exists.

Unfortunately it is not clear what to do with recursive functions in the measure-term translation. Since it is not practical to build and evaluate infinite measure terms, some sort of approximation is necessary. Pfeffer and Koller (2000) presents an approximation technique over *probabilistic relational knowledge bases*, which can be expressed as infinite measure terms. It remains to be seen how to extend such techniques to arbitrary measure terms and how to approximate expressions involving higher-order and recursive functions that produce measure terms.

Although our presentation of the probability monad is general, as are our laws for `sample` and `expectation`, much of the work in this paper is focused on discrete distributions with finite support. The moment we introduce recursive functions, however, we can use the probability monad to define distributions with *uncountable* support. For example, there are uncountably many infinite sequences of tosses of a fair coin. If we have a lazy variant of a stochastic lambda calculus, or if we embed the probability monad in Haskell, it is perfectly reasonable to write functions that produce infinite data structures and to make queries over them. Queries that depend only on finite sub-structures (e.g., the the probability that the first ten tosses of a coin come out heads) should be computable in finite time using lazy evaluation. We would like to extend our monadic implementation to incorporate such infinite models and to answer such queries effectively. The natural measurable space over which such queries would make sense should be the Borel sets of the Scott topology.

In the long term, we would like to explore *Bayesian learning* in a monadic framework. In the Bayesian paradigm, the parameters p passed to *choose* are not known exactly, but are themselves defined only by a probability distribution. A Bayesian experiment consists of a model, a *prior* distribution over parameters p, and a set of observations; the result is a *posterior* distribution over the parameters. For example, we could use Bayesian experiments to estimate the probability that an aggressive driver runs a red light. To incorporate the Bayesian approach into our monadic framework, we would need to make the parameter p an expression, not a value. Such an extension might provide a useful declarative foundation for an active area of machine learning.

Acknowledgments

Simon Peyton Jones helped unstick our discussions of measure terms and variable elimination. Bob Muller helped us work out *lift* and guided us through the thickets of domain theory. We enjoyed many stimulating discussions with members of the CS 252 seminar, especially Chung-chieh Shan and Dylan Thurston. Jon Eddy, Simon Peyton Jones, and Chung-chieh Shan provided helpful comments on the manuscript. The anonymous referees not only directed us to useful related work but also made suggestions that helped us improve the paper significantly.

This work was supported by NSF grant CCR-0096069, by an Alfred P. Sloan Research Fellowship, and by Harvard University.

References

Appel, Andrew W. 1992. *Compiling with Continuations.* Cambridge: Cambridge University Press.

Arnborg, Stefan. 1985. Efficient algorithms for combinatorial problems on graphs with bounded decomposability. *BIT*, 25(1):2–23.

Benveniste, Albert, Bernard C. Levy, Eric Fabre, and Paul Le Guernic. 1995. A calculus of stochastic systems for the specification, simulation, and hidden state estimation of mixed stochastic/nonstochastic systems. *Theoretical Computer Science*, 152(2):171–217.

Charniak, Eugene. 1993. *Statistical Language Learning.* MIT Press.

Claessen, Koen and John Hughes. 2000 (September). QuickCheck: a lightweight tool for random testing of Haskell programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00),* in *SIGPLAN Notices*, 35(9):268–279.

Danvy, Olivier. 1996. Type-directed partial evaluation. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, New York, NY.

Dechter, Rina. 1996 (August). Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI-96)*, pages 211–219, San Francisco.

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

Gupta, Vineet, Radha Jagadeesan, and Prakash Panangaden. 1999 (January). Stochastic processes as concurrent constraint programs. In *Conference Record of the 26th Annual ACM Symposium on Principles of Programming Languages*, pages 189–202.

Hughes, John. 1989 (April). Why functional programming matters. *The Computer Journal*, 32(2):98–107.

———. 1995. The design of a pretty-printing library. In Jeuring, Johan and Erik Meijer, editors, *Advanced Functional Programming*, Vol. 925 of *Lecture Notes in Computer Science*. Springer Verlag.

Jaakkola, Tommi S. and Michael I. Jordan. 1999. Variational probabilistic inference and the QMR-DT network. *Journal of Artificial Intelligence Research*, 10:291–322.

Jensen, Finn V. 1996. *An Introduction to Bayesian Networks.* New York: Springer.

Jones, Claire. 1990 (July). *Probabilistic Non-determinism.* PhD thesis, Department of Computer Science, University of Edinburgh. Also Laboratory for the Foundations of Computer Science technical report ECS-LFCS-90-105. Available online.

Jones, Claire and Gordon D. Plotkin. 1989. A probabilistic powerdomain of evaluations. In *Proceedings of the Fourth Annual IEEE Symposium On Logic In Computer Science*, pages 186–195.

Jordan, Michael I., editor. 1998. *Learning in Graphical Models.* Kluwer.

Koller, Daphne, David McAllester, and Avi Pfeffer. 1997. Effective Bayesian inference for stochastic programs. In *Fourteenth National Conference on Artificial Intelligence (AAAI)*, pages 740–747.

Lauritzen, Steffen L. and David J. Spiegelhalter. 1988. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, pages 157–224.

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

Li, Zhaoyu and Bruce d'Ambrosio. 1994. Efficient inference in Bayes nets as a combinatorial optimization problem. *International Journal of Approximate Reasoning*, 11(1):55–81.

Luger, George and Dan Pless. 2001. A stochastic λ-calculus. Technical Report TR-CS-2001-04, Department of Computer Science, University of New Mexico.

Mahoney, Suzanne M. and Kathryn Blackmond Laskey. 1998 (July). Constructing situation specific belief networks. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 370–378. Morgan Kaufmann.

Muggleton, Stephen. 2001. Stochastic logic programs. *Journal of Logic Programming.* Accepted subject to revision.

Pearl, Judea. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA: Morgan Kaufmann.

Pfeffer, Avi. 2001 (August). IBAL: A probabilistic rational programming language. In *Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 733–740, Seattle.

Pfeffer, Avi and Daphne Koller. 2000 (July). Semantics and inference for recursive probability models. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00)*, pages 538–544, Menlo Park, CA.

Ramsey, Norman. 1994 (September). Literate programming simplified. *IEEE Software*, 11(5):97–105.

Rudin, Walter. 1974. *Real and Complex Analysis*. Series in Higher Mathematics. Second edition. New York: McGraw-Hill.

Saheb-Djahromi, N. 1978 (September). Probabilistic LCF. In Winkowski, Józef, editor, *Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science*, Vol. 64 of *Lecture Notes in Computer Science*, pages 442–451. Springer.

———. 1980 (September). CPO's of measures for nondeterminism. *Theoretical Computer Science*, 12(1):19–37.

Smyth, Michael B. 1983 (July). Power domains and predicate transformers: A topological view. In Díaz, Josep, editor, *Automata, Languages and Programming, 10th Colloquium (ICALP-83)*, Vol. 154 of *Lecture Notes in Computer Science*, pages 662–675, Barcelona, Spain.

Wadler, Philip. 1992 (January). The essence of functional programming (invited talk). In *Conference Record of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 1–14. New York, NY: ACM Press.

Witten, Ian H., Radford M. Neal, and John G. Cleary. 1987 (June). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.

Zhang, Nevin L. and David Poole. 1994. A simple approach to Bayesian network computations. In *Tenth Biennial Canadian Artificial Intelligence Conference*.

A. Implementations of probability monads

These implementations are derived from the algebraic laws in Section 5, using techniques explained by Hughes (1989). This paper was prepared using the Noweb system for "literate programming" (Ramsey 1994), and all the Haskell code in the paper has been automatically extracted and run through the Glasgow Haskell Compiler, version 4.06.

The support monad In the support monad, we represent a distribution by a list of values it could contain.

⟨*probability monads*⟩+≡
```
newtype Support a = Support [a]
instance Monad Support where
  return x = Support [x]
  (Support l) >>= k =
    Support (concat [s | x <- l, let Support s = k x])
instance ProbabilityMonad Support where
  choose _ (Support l) (Support l') = Support (l ++ l')
instance SupportMonad Support where
  support (Support l) = l
```

The expectation monad We represent the expectation monad by a function that computes expectation directly.

⟨*probability monads*⟩+≡
```
newtype Exp a = Exp ((a -> Double) -> Double)
instance Monad Exp where
  return x = Exp (\h -> h x)
  (Exp d) >>= k =
     Exp (\h -> let apply (Exp f) arg = f arg
                    g x = apply (k x) h
                in  d g)
instance ProbabilityMonad Exp where
  choose p (Exp d1) (Exp d2) =
    Exp (\h -> p * d1 h + (1-p) * d2 h)
instance ExpMonad Exp where
  expectation h (Exp d) = d h
```

The sampling monad Again, we represent the sampling monad as a suitable function.

⟨*probability monads*⟩+≡
```
newtype Sample a = Sample (RandomGen g => g -> (a, g))
instance Monad Sample where
  return x = Sample (\ g -> (x, g))
  (Sample s) >>= k =
      Sample (\ g -> let (a, g')   = s g
                         Sample s' = k a
                     in  s' g')
instance ProbabilityMonad Sample where
  choose p (Sample s1) (Sample s2) =
    Sample (\g -> let (x, g') = Random.random g
                  in  (if x < p then s1 else s2) g')
instance SamplingMonad Sample where
  sample (Sample s) g = s g
```

Addendum

We overlooked at least two relevant pieces of related work. Andrejz Filinski's doctoral dissertation presents something about the probability monad. David Monniaux published something relevant in ESOP'01.