

A Monadic Probabilistic Language

Sungwoo Park, Frank Pfenning, Sebastian Thrun
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
USA
{gla,fp,thrun}@cs.cmu.edu

ABSTRACT

Motivated by many practical applications that have to compute in the presence of uncertainty, we propose a monadic probabilistic language based upon the mathematical notion of *sampling function*. Our language provides a unified representation scheme for probability distributions, enjoys rich expressiveness, and offers high versatility in encoding probability distributions. We also develop a novel style of operational semantics called a *horizontal operational semantics*, under which an evaluation returns not a single outcome but multiple outcomes. We have preliminary evidence that the horizontal operational semantics improves the ordinary operational semantics with respect to both execution time and accuracy in representing probability distributions.

Categories and Subject Descriptors

D.3.1 [Formal Definitions and Theory]: Semantics, Syntax

General Terms

Languages

Keywords

Probabilistic language, Monad

1. INTRODUCTION

In recent years, a number of scientific disciplines have embarked on probabilistic techniques for information processing. Probabilistic computation forms the core of today's best speech recognizers [28, 36, 16], natural language processing systems [3], biological sequence analyzers [14, 21], time series predictors [12], computer vision systems [15, 4], and robotic systems [33, 35]. All these fields share the characteristic that sensors are noisy. To accommodate uncertainty in sensor data, information is represented by probability distributions.

As probabilistic computation plays an increasing role in diverse fields, researchers have also designed *probabilistic languages*, which incorporate probabilistic computation as an inherent component. In particular, there have been efforts in the design of probabilistic languages to employ *monads*, which have become popular as a means of formulating the semantics of computation [22, 23]. Ramsey and Pfeffer [29] present a stochastic lambda calculus whose denotational semantics is based upon the monad of probability measures [10]. Jones [17] presents a probabilistic metalanguage based upon the monad of evaluations [18]. Both monads do not distinguish discrete and continuous probability distributions, and provide a unified representation scheme for all kinds of probability distributions.

The above languages, however, do not fully implement their monads. The difficulty is that a probability measure or an evaluation conceptually maps not observations but events (sets of observations) to probabilities. For instance, a probability measure on a measurable space Ω is a mapping from not Ω but its σ -algebra \mathcal{B}_Ω to $[0.0, 1.0]$. Since it is computationally infeasible to keep track of probabilities assigned to all events from an infinite discrete domain or a continuous domain, they provide only a binary choice construct (e_1 or_p e_2 in [18] and **choose** p e_1 e_2 in [29]). As a result, both languages essentially implement the probability monad in [6], which is capable of specifying only probability distributions over finite domains.

In this paper, we propose a monadic probabilistic language capable of specifying probability distributions over infinite discrete domains and continuous domains. The mathematical basis for our language is *sampling functions*, which map the unit interval $(0.0, 1.0]$ to probability domains. We build a monad based upon sampling functions, and employ it as the probability monad for our language. For the syntax for our language, we employ the new formulation of the computational λ -calculus in [26], which draws a syntactic distinction between values and computations. Our language provides a unified representation scheme for probability distributions, enjoys rich expressiveness, and offers high versatility in encoding probability distributions.

We also investigate how to implement the expectation query on probability distributions in our language. Although our language does not permit a precise implementation of the expectation query, we use the Monte Carlo method [8] as a practical solution. Under an ordinary operational semantics, we can implement the expectation query through multiple independent evaluations, all of which repeat the same computation. In order to achieve an efficient imple-

mentation, we develop a novel style of operational semantics called a *horizontal operational semantics*, under which a single evaluation returns multiple outcomes. Experimental results show that the horizontal operational semantics consistently outperforms the ordinary operational semantics in seven test cases with respect to both accuracy and execution time.

The rest of this paper is organized as follows. In Section 2, we give an overview of the language. In Section 3, we present our monadic probabilistic language λ_{\circ} with its type system and operational semantics. In Section 4, we demonstrate the expressive power of λ_{\circ} by implementing various probability distributions. In Section 5, we present a horizontal operational semantics. In Section 6, we discuss how to implement the horizontal operational semantics. In Section 7, we present experimental results. In Section 8, we discuss related work. In Section 9, we conclude with future work.

2. OVERVIEW

2.1 Sampling monad

The expressive power of a probabilistic language is determined to a large extent by its mathematical foundation. For instance, in a probabilistic language based upon probability mass functions, it is hard to specify continuous probability distributions in terms of the primitive construct provided by the language. Motivated by the kinds of probabilistic computations mentioned in the introduction, we choose *sampling functions* as our mathematical basis. A sampling function is a mapping from the unit interval $(0.0, 1.0]$ to a probability domain. Given a random number generated from a uniform distribution $U(0.0, 1.0]$ over the unit interval, it returns an observation, or a *sample*, from the probability domain, and thus specifies a unique probability distribution. As demonstrated by the first author [24], the use of sampling functions leads to a unified representation scheme for probability distributions, rich expressiveness, and high versatility in a probabilistic language.

Since a typical characterization of a probability distribution states that it consumes zero or more random numbers, we redefine the notion of sampling function as follows: a sampling function takes as input an infinite sequence of random numbers generated from $U(0.0, 1.0]$, consumes as many random numbers as it needs (possibly none), and returns a sample together with the remaining sequence of random numbers. Then, from the new definition of sampling functions, we obtain a state monad \circ such that $\circ A = (A \times S)^S$ where A is a type and S is the set of states, namely the set of infinite sequences of random numbers generated from $U(0.0, 1.0]$. We refer to the state monad \circ as a *sampling monad* and employ it as the probability monad for our language.

In order to create probability distributions other than point mass distributions in the sampling monad \circ , we need an operation for consuming random numbers in addition to the standard unit and bind operations. For our purpose, an operation that consumes only the first random number in a given state is sufficient because, in conjunction with the bind operation, it enables us to consume as many random numbers as we need.

2.2 Syntax and operational semantics

Moggi [22, 23] proposes a monadic metalanguage as a foundation for the semantics of programming languages that distinguish between values and computations. Benton, Biermann, and de Paiva [1] show that it also arises as the term calculus corresponding to lax logic [5] via the Curry-Howard isomorphism.

From a logical perspective, the monadic meta-language is complete with respect to normalization because it includes not only β reductions but also a commuting conversion. In the case of lax logic presented in [1], we need the commuting conversion because of the elimination rule for the lax modality \circ :

$$\frac{\begin{array}{c} [A] \\ \dots \\ \circ A \quad \circ C \end{array}}{\circ C} (\circ_{\varepsilon})$$

The elimination rule is not pure in the sense that in order to eliminate $\circ A$ we must analyze simultaneously another instance of \circ , namely $\circ C$.

Pfenning and Davies [26] present a new formulation of the computational λ -calculus that does not require the commuting conversion. The new formulation draws a syntactic distinction between values and computations through the use of two syntactic categories: *terms* for values and *expressions* for computations. By the Curry-Howard isomorphism, the syntactic distinction corresponds to the use of two forms of judgments in the new formulation of lax logic: judgments for truth, whose evidence is terms, and judgments for possible necessity, whose evidence is expressions.

Harper and Pfenning [13] present an operational semantics for the new formulation of the computational λ -calculus. It is based upon the *possible world interpretation*: the evaluation of an expression takes a world as input, performs a computation, and returns a value together with a resultant world. The operational semantics is suitable for our language because states in the sampling monad \circ are simply a particular class of worlds. We therefore base the syntax and the operational semantics of our language upon the new formulation of the computational λ -calculus, using terms for values and expressions for probabilistic computations. Our examples can be translated into more traditional monadic style as in Haskell in a relatively straightforward manner.

2.3 Horizontal operational semantics

Among common queries on probability distributions, the expectation query is the most important. The expectation of a function f with respect to a probability distribution μ is a definite integral $\int f d\mu$, *i.e.* the mean of f over μ [30]. Other queries may be derived as special cases of the expectation query. For instance, the mean of a probability distribution over real numbers is the expectation of an identity function.

Our language, however, does not permit a precise implementation of the expectation query. Intuitively we cannot convert an arbitrary sampling function to an equivalent probability measure. A practical solution is to use the Monte Carlo method, which states that we can estimate $\int f d\mu$ by generating a large number of samples X_i from μ and computing the mean of $f(X_i)$:

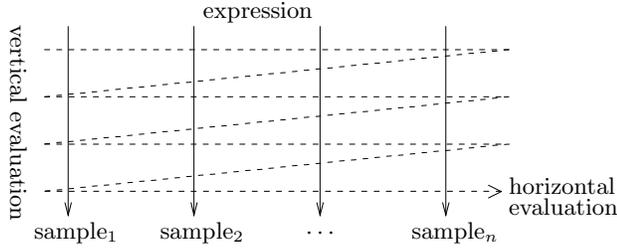
$$\lim_{r \rightarrow \infty} \frac{f(X_1) + \dots + f(X_r)}{r} = \int f d\mu$$

Given an expression, we can generate a set of samples

type	A	$::=$	$P \mid A \supset A \mid \circ A$
base type	P	$::=$	\mathbf{real}
term	M, N	$::=$	$x \mid \lambda x:A. M \mid M M \mid$ $\mathbf{prob} E \mid \mathbf{fix} x:A. M \mid u$
expression	E, F	$::=$	$\mathbf{sample} x \triangleleft M \text{ in } E \mid M \mid \mathcal{U}$
value	V, W	$::=$	$\lambda x:A. M \mid \mathbf{prob} E \mid u$
random number	u	\in	$(0.0, 1.0]$
sampling sequence	s	$::=$	$\epsilon \mid us$
typing context	Γ	$::=$	$\cdot \mid \Gamma, x : A$

Figure 1: Abstract syntax for λ_0 .

by evaluating it repeatedly, *i.e.*, through multiple *vertical evaluations*. This refers to the primary specification of our language, as given in Section 3.3. Alternatively, we can obtain a set of samples through a single *horizontal evaluation*, which, conceptually, performs multiple vertical evaluations simultaneously:



We refer to horizontal evaluation rules for expressions as a *horizontal operational semantics*. A horizontal semantics is by no means uniquely determined from a vertical one—many choices and implementation decision still remain. However, as our results show and experience from realistic applications suggest (see, for example, [8, 7]), it is often more efficient and robust in practice.

After investigating the vertical semantics, we also develop a framework for horizontal implementations and prove that it respects the vertical semantics in a manner made precise in Section 5.

3. LANGUAGE λ_0

3.1 Abstract syntax

We obtain our monadic probabilistic language λ_0 by extending the new formulation of the computational λ -calculus in [26] (Figure 1). A *probability term* $\mathbf{prob} E$ encapsulates an expression E by converting it into a value. $\mathbf{fix} x:A. M$ is a fixed point construct for terms. There are three kinds of expressions: a *sampling expression* $\mathbf{sample} x \triangleleft M \text{ in } E$, a *term expression* M , and a *uniform expression* \mathcal{U} . A random number u comes from the evaluation of a uniform expression and is not part of the source language.

A *sampling sequence* s consists of random numbers generated from $U(0.0, 1.0]$ by the runtime system. ϵ denotes an empty sampling sequence. We write $s_1 \cdot s_2$ for the concatenation of two sampling sequences s_1 and s_2 .

3.2 Type system

The type system employs a term typing judgment of the form $\Gamma \vdash M : A$ and an expression typing judgment of the form $\Gamma \vdash E \div A$, where a typing context Γ is a set of bindings of the form $x : A$ (Figure 2). $\Gamma \vdash M : A$ means that M

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \mathbf{tt_var} \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x:A. M : A \supset B} \mathbf{tt_lam}$$

$$\frac{\Gamma \vdash M_1 : B \supset A \quad \Gamma \vdash M_2 : B}{\Gamma \vdash M_1 M_2 : A} \mathbf{tt_app}$$

$$\frac{\Gamma \vdash E \div A}{\Gamma \vdash \mathbf{prob} E : \circ A} \mathbf{tt_prob}$$

$$\frac{\Gamma, x : A \vdash M : A}{\Gamma \vdash \mathbf{fix} x:A. M : A} \mathbf{tt_mfix} \quad \frac{}{\Gamma \vdash u : \mathbf{real}} \mathbf{tt_random}$$

$$\frac{\Gamma \vdash M : \circ A \quad \Gamma, x : A \vdash E \div B}{\Gamma \vdash \mathbf{sample} x \triangleleft M \text{ in } E \div B} \mathbf{et_sample}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M \div A} \mathbf{et_term} \quad \frac{}{\Gamma \vdash \mathcal{U} \div \mathbf{real}} \mathbf{et_U}$$

Figure 2: Typing rules for λ_0 .

$$\frac{}{\lambda x:A. M \hookrightarrow \lambda x:A. M} \mathbf{te_lam}$$

$$\frac{M_1 \hookrightarrow \lambda x:A. M \quad [M_2/x]M \hookrightarrow V}{M_1 M_2 \hookrightarrow V} \mathbf{te_app}$$

$$\frac{}{\mathbf{prob} E \hookrightarrow \mathbf{prob} E} \mathbf{te_prob}$$

$$\frac{[\mathbf{fix} x:A. M/x]M \hookrightarrow V}{\mathbf{fix} x:A. M \hookrightarrow V} \mathbf{te_mfix} \quad \frac{}{u \hookrightarrow u} \mathbf{te_random}$$

Figure 3: Term evaluation rules for λ_0 .

denotes a value of type A under the typing context Γ , and $\Gamma \vdash E \div A$ means that E describes a probabilistic computation of type A under the typing context Γ .

3.3 Operational semantics

We write $[N/x]M$ and $[N/x]E$ for the capture-avoiding substitution of N for the variable x in M and E , respectively. Note that we bind a variable x in both a lambda abstraction $\lambda x:A. M$ and a sampling expression $\mathbf{sample} x \triangleleft M \text{ in } E$.

The operational semantics employs a term evaluation judgment of the form $M \hookrightarrow V$ (Figure 3) and an expression evaluation judgment of the form $E \xrightarrow{s} V$ (Figure 4). $E \xrightarrow{s} V$ means that the expression E evaluates to a value V by consuming a sampling sequence s . By the rule $\mathbf{ee_U}$, the runtime system generates a random number u from $U(0.0, 1.0]$, and substitutes it for the uniform expression \mathcal{U} .

In terms of the sampling monad \circ , in which a state is an infinite sequence of random numbers generated from $U(0.0, 1.0]$, $E \xrightarrow{s} V$ means that the expression E takes a state w as input, consumes a prefix s of w , and returns a sample V with the resultant state w' . Under the possible world interpretation in [13], the sampling sequence s is the difference between the two worlds w and w' , and serves as evidence of a pre-order relation from w to w' . Hence the rule $\mathbf{ee_sample}$ implements the bind operation, and the rule $\mathbf{ee_term}$ the unit operation. The rule $\mathbf{ee_U}$ simply consumes the first random number in

$$\begin{array}{c}
\frac{M \hookrightarrow \text{prob } F \quad F \xrightarrow{s} W \quad [W/x]E \xrightarrow{s'} V}{\text{sample } x \triangleleft M \text{ in } E \xrightarrow{s \circ s'} V} \text{ ee_sample} \\
\\
\frac{M \hookrightarrow V}{M \xrightarrow{\epsilon} V} \text{ ee_term} \quad \frac{u \sim U(0.0, 1.0)}{\mathcal{U} \xrightarrow{u\epsilon} u} \text{ ee_U}
\end{array}$$

Figure 4: Expression evaluation rules for λ_o .

a given state.

The uniqueness and type preservation properties are stated as follows:

THEOREM 3.1 (UNIQUENESS). *If $\cdot \vdash M : A$, then there exists at most one value V such that $M \hookrightarrow V$. If $\cdot \vdash E \div A$, then for any sampling sequence s , there exists at most one sample W such that $E \xrightarrow{s} W$.*

PROOF. By induction over the structure of $M \hookrightarrow V$ and $E \xrightarrow{s} W$. We use the fact that if $E \xrightarrow{s} W$ and $E \xrightarrow{s \circ s'} W'$, then $s' = \epsilon$ and $W = W'$. \square

THEOREM 3.2 (TYPE PRESERVATION). *If $M \hookrightarrow V$ and $\cdot \vdash M : A$, then $\cdot \vdash V : A$. If $E \xrightarrow{s} W$ and $\cdot \vdash E \div A$, then $\cdot \vdash W : A$.*

PROOF. By simultaneous induction over the structure of $M \hookrightarrow V$ and $E \xrightarrow{s} W$, using a straightforward substitution lemma. \square

With a small-step semantics we could also prove an appropriate progress theorem. However, since the implementation is not directly based on this (vertical) semantic specification, we forego the routine details.

3.4 Joint distributions

In order to support joint distributions, we introduce product terms:

$$\begin{array}{l}
\text{type } A ::= \dots \mid A \times A \\
\text{term } M ::= \dots \mid (M, M) \mid \text{fst } M \mid \text{snd } M \\
\text{value } V ::= \dots \mid (M, M)
\end{array}$$

The typing rules and evaluation rules are standard. Then we can simulate the joint distribution $M_A \star M_B : \circ(A \times B)$ between two probability distributions $M_A : \circ A$ and $M_B : \circ B$ with:

$$\text{prob sample } x_A \triangleleft M_A \text{ in sample } x_B \triangleleft M_B \text{ in } (x_A, x_B)$$

3.5 Conditional probabilities

A conditional probability takes a value and returns a probability distribution. We write $\circ B_{|A}$ for the type of a conditional probability that takes a value of type A and returns a probability distribution of type $\circ B$. We create a conditional probability of type $\circ B_{|A}$ with a *conditional probability construct* $\oint x : A. E$, which is syntactic sugar for $\lambda x : A. \text{prob } E$. Hence $\circ B_{|A}$ is defined as $A \supset \circ B$. We can simulate the integration $M_1 \bullet M_2$ between a conditional probability M_1 and a probability distribution M_2 with:

$$\text{prob sample } x \triangleleft M_2 \text{ in sample } y \triangleleft M_1 \text{ in } y$$

Then we derive the typing rules for $\oint x : A. E$ and $M_1 \bullet M_2$:

$$\frac{\Gamma, x : A \vdash E \div B}{\Gamma \vdash \oint x : A. E : \circ B_{|A}} \text{ tt_cnd} \quad \frac{\Gamma \vdash M_1 : \circ B_{|A} \quad \Gamma \vdash M_2 : \circ A}{\Gamma \vdash M_1 \bullet M_2 : \circ B} \text{ tt_itg}$$

3.6 A fixed point construct for expressions

In λ_o , expressions describe non-recursive probabilistic computations. Since some probability distributions are defined in a recursive way (e.g., a geometric distribution), it is desirable to be able to describe recursive probabilistic computations as well. To this end, we introduce an *expression variable* \mathbf{x} and an *expression fixed point construct* $\text{efix } \mathbf{x} \div A. E$:

$$\text{expression } E ::= \dots \mid \mathbf{x} \mid \text{efix } \mathbf{x} \div A. E$$

We also introduce a new form of binding $\mathbf{x} \div A$:

$$\text{typing context } \Gamma ::= \dots \mid \Gamma, \mathbf{x} \div A$$

Then the new expression typing rules are:

$$\frac{\mathbf{x} \div A \in \Gamma}{\Gamma \vdash \mathbf{x} \div A} \text{ et_var} \quad \frac{\Gamma, \mathbf{x} \div A \vdash E \div A}{\Gamma \vdash \text{efix } \mathbf{x} \div A. E \div A} \text{ te_efix}$$

We write $[F/\mathbf{x}]M$ and $[F/\mathbf{x}]E$ for the capture-avoiding substitution of F for the expression variable \mathbf{x} in M and E , respectively. We evaluate $\text{efix } \mathbf{x} \div A. E$ by the following rule:

$$\frac{[\text{efix } \mathbf{x} \div A. E/\mathbf{x}]E \xrightarrow{s} V}{\text{efix } \mathbf{x} \div A. E \xrightarrow{s} V} \text{ ee_efix}$$

Simulating the expression fixed point construct

Since a probability term enables us to encapsulate a probabilistic computation, we can simulate $\text{efix } \mathbf{x} \div A. E$ with a fixed point construct for terms. The basic idea is that we define a translation function $(\cdot)^*$ such that $(\text{efix } \mathbf{x} \div A. E)^*$ is given by:

$$\text{sample } x_r \triangleleft \text{fix } x_p : \circ A. \text{prob} [\text{sample } x_v \triangleleft x_p \text{ in } x_v/\mathbf{x}]E^* \text{ in } x_r$$

In other words, we introduce a recursive term x_p to encapsulate a recursive expression $\text{efix } \mathbf{x} \div A. E$ and expand \mathbf{x} to a sampling expression $\text{sample } x_v \triangleleft x_p \text{ in } x_v$. The translation of all the other forms of terms and expressions is structural.

With the translation function $(\cdot)^*$, we can show that the typing rules et_var and te_efix are sound in the original definition of λ_o . The following theorem shows that the evaluation rule ee_efix is also sound. Note that for any expression E , the evaluation of E^* does not require the rule ee_efix .

THEOREM 3.3. *$M \hookrightarrow V$ if and only if $M^* \hookrightarrow V^*$, and $E \xrightarrow{s} W$ with the rule ee_efix if and only if $E^* \xrightarrow{s} W^*$.*

PROOF. See Appendix. \square

Now we can use $\text{efix } \mathbf{x} \div A. E$ as a derived form of expression in the original definition of λ_o . In the next section, we demonstrate the expressive power of λ_o by implementing various probability distributions.

4. EXAMPLES

The first author has previously presented a probabilistic calculus whose mathematical basis is sampling functions [24], but without monadic encapsulation. This calculus has three desirable properties. First it provides a unified representation scheme for probability distributions: we no longer distinguish between discrete probability distributions, continuous probability distributions, and even probability distributions that do not belong to either group. Second it enjoys rich expressiveness: we can specify probability distributions over infinite discrete domains, continuous domains, and even unusual domains such as infinite data structures

(*e.g.*, trees) and cyclic domains (*e.g.*, angular values). Third it enjoys high versatility: there can be more than one way to specify a probability distribution, and the more we know about it, the better we can encode it.

Since the monadic language of this paper is also founded upon sampling functions, it also enjoys the three properties above. Probability terms further enhance its expressiveness because we can encapsulate probabilistic computations and treat probability distributions as values. For instance, we can now specify a probability distribution over probability distributions of the same type. The examples below demonstrate the expressive power of our language. See [2] for the simulation methods used in the examples.

We first introduce new base types and primitive constructs such as `if M then M else M`. `let x = M in N` is syntactic sugar for $(\lambda x:A.N) M$, and `let rec x = M in N` for `let x = fix x:A.M in N`, where A is an appropriate type for M . `unprob M` is syntactic sugar for `sample x < M in x`. It retrieves a probabilistic computation from the probability distribution M . `if M then E1 else E2` is syntactic sugar for `unprob (if M then prob E1 else prob E2)`. It branches to either E_1 or E_2 depending on the result of evaluating M .

As a simple example, we can encode a uniform distribution over a real interval $(a, b]$ by exploiting the definition of the uniform operation:

```
let uniform = λa:real. λb:real.
  prob sample x < prob U in a + x * (b - a)
```

In a similar way, we can encode a Bernoulli distribution over type `bool` with parameter p :

```
let bernoulli = λp:real. prob sample x < prob U in x ≤ p
```

We can encode a binomial distribution with parameters p and n_0 by exploiting probability terms:

```
let binomial = λp:real. λn0:int.
  let bernoullip = bernoulli p in
  let rec binomialp = λn:int.
    if n = 0 then prob 0
    else prob sample x < binomialp (n - 1) in
      sample b < bernoullip in
        if b then 1 + x else x
  in
  binomialp n0
```

Here `binomialp` takes an integer n as input and returns a binomial distribution with parameters p and n .

If a probability distribution is defined in terms of a recursive process of generating samples, we can encode it directly with a recursive term or expression. For instance, we can encode a geometric distribution with parameter p as follows:

```
let geometric_rec = λp:real.
  let bernoullip = bernoulli p in
  let rec geometric =
    prob sample b < bernoullip in
      eif b then 0
    else
      sample x < geometric in
        1 + x
  in
  geometric
```

Here we use a recursive term `geometric` of type `oint`. Equiv-

alently we can use a recursive expression:

```
let geometric_fix = λp:real.
  let bernoullip = bernoulli p in
  prob eif geometric ÷ int.
    sample b < bernoullip in
      eif b then 0
    else
      sample x < prob geometric in
        1 + x
```

Another way to encode a geometric distribution is by employing the inverse of its cumulative distribution function as a sampling function, which is known as the inverse transform method:

```
let geometric_inverse = λp:real.
  prob sample x < U in 1 + ⌊log x / log (1.0 - p)⌋
```

The rejection method, which enables us to generate a sample from a probability distribution by repeatedly generating samples from other probability distributions until they satisfy a certain condition, can be implemented with recursive terms or expressions. For instance, we can encode a Gaussian distribution with mean m and variance σ^2 by the rejection method with respect to exponential distributions:

```
let bernoulli0.5 = bernoulli 0.5
let exponential1.0 = prob sample x < U in -log x
let gaussian_rejection = λm:real. λσ:real.
  prob eif gaussian ÷ real.
    sample y1 < exponential1.0 in
    sample y2 < exponential1.0 in
    eif y2 ≥ (y1 - 1.0)2 / 2.0 then
      sample b < bernoulli0.5 in
        if b then m + σ * y1 else m - σ * y1
    else
      gaussian
```

Here `exponential1.0` encodes an exponential distribution with parameter 1.0 by the inverse transform method.

The evaluation of `gaussian` consumes at least three random numbers. We can encode a Gaussian distribution with only two random numbers:

```
let gaussian_BoxMuller = λm:real. λσ:real.
  prob sample u < prob U in
    sample v < prob U in
      m + σ * √(-2.0 * log u * cos (2.0 * π * v))
```

We can also approximate a Gaussian distribution by exploiting the central limit theorem (although usually one would choose a larger number of samples):

```
let gaussian_central = λm:real. λσ:real.
  prob sample s1 < prob U in
    sample s2 < prob U in
    sample s3 < prob U in
      m + σ * (2.0 * (s1 + s2 + s3) - 3.0)
```

Assuming that type constructors, pattern matching on product terms, and list constructors (`[]` and `:`) are available,

we can implement a coin-flip hidden Markov model [28]:

```

data output = Head | Tail
type state = ◦(output × state)
let rec (coin1, coin2) : state × state =
  ((bernoullioutput 0.75) ★ (bernoullistate 0.5 coin1 coin2),
   (bernoullioutput 0.25) ★ (bernoullistate 0.5 coin1 coin2))
let rec experiment = λn: int. §(output, next): output × state.
  eif n = 0 then []
  else
    sample rest ◁ experiment (n - 1) • next in
    output : rest
let initial_state = bernoullistate 0.5 coin1 coin2
let result = experiment n0 • initial_state

```

Here we assume that $bernoulli_{output}$ and $bernoulli_{state}$ implement Bernoulli distributions over types `output` and `state`, respectively. `result` denotes the probability distribution of the first n_0 outcomes from the above hidden Markov model.

As a final example, we can implement a belief network [31] by exploiting conditional probability constructs:

```

let alarm = §(burglary, earthquake): bool × bool.
  eif burglary then unprob Palarm|burglary
  else eif earthquake then unprob Palarm|¬burglary∧earthquake
  else unprob Palarm|¬burglary∧¬earthquake
let john_calls = §alarm: bool.
  eif alarm then unprob Pjohn_calls|alarm
  else unprob Pjohn_calls|¬alarm

```

$P_{alarm|burglary}$ denotes the probability distribution that the alarm goes off when a burglary happens (similarly for other variables of the form $P_{\cdot|\cdot}$). The conditional probabilities `alarm` and `john_calls` do not answer any query on the belief network; they only describe its structure. In order to answer a specific query on it, we usually implement a corresponding probability distribution. For instance, we can use $Q_{burglary|john_calls}$ below to answer “what is the probability $P_{burglary|john_calls}$ that a burglary has happened when John calls?”:

```

let Qburglary|john_calls =
  prob efix z ÷ bool.
    sample b ◁ Pburglary in
    sample j ◁
      john_calls • (alarm • (prob b ★ Pearthquake)) in
    eif j then b else z

```

$P_{burglary}$ denotes the probability distribution that a burglary happens, and $P_{earthquake}$ that an earthquake happens. The probability $P_{burglary|john_calls}$ is equal to the expectation of a function f with respect to $Q_{burglary|john_calls}$ where $f(\text{True}) = 1.0$ and $f(\text{False}) = 0.0$. Thus we can compute it with the expectation query on probability distributions. In the next section, we investigate how to implement the expectation query in our language.

5. HORIZONTAL OPERATIONAL SEMANTICS

5.1 Expectation query

In our language¹ it is easy to implement a polymorphic function that computes a *probabilistic* expectation

$$P_{expectation_A} : (A \triangleright \text{real}) \triangleright \circ A \triangleright \text{real}.$$

¹extended with polymorphism in a standard way

$$s :: Z = \{(s \cdot s_i, V_i) \mid (s_i, V_i) \in Z\}$$

$$\frac{M \hookrightarrow \text{prob } F \quad F \rightsquigarrow \{(s_i, V_i) \mid i\} \quad [V_i/x]E \rightsquigarrow Z_i}{\text{sample } x \triangleleft M \text{ in } E \rightsquigarrow \bigcup_i s_i :: Z_i} \text{ hee_sample}$$

$$\frac{M \hookrightarrow V}{M \rightsquigarrow \{(\epsilon, V), \dots, (\epsilon, V)\}} \text{ hee_term} \quad \frac{u_i \sim U(0.0, 1.0)}{\mathcal{U} \rightsquigarrow \{(u_i \epsilon, u_i) \mid i\}} \text{ hee_U}$$

Figure 5: Horizontal evaluation rules

When given a function f and a term M denoting a probability distribution μ , $P_{expectation} f M$ estimates the expectation of f with respect to μ by generating a large number of samples. The output is probabilistic, because it depends on these samples.

While semantically clean, such a function has the drawback that any further computation that depends on the expectation will have to be encapsulated in the monad. To avoid this there are several choices. For example, we could provide an analogue to `performIO` in Haskell which is unsafe in some sense, but might work in practice. That is, we could provide the programmer with the function `performMean` : `oreal` \triangleright `real` that approximates the mean and can be composed with the above function. Unfortunately, this violates the separation of effectful and effect-free computations maintained by the monad.

Instead we consider in this section how to provide a framework in which a construct

$$expectation_A : (A \triangleright \text{real}) \triangleright \circ A \triangleright \text{real}$$

would be sound because the implementation guarantees a unique answer, even if it may only be approximate. We plan to explore the other possibilities in future research.

Therefore we develop a framework for a family of implementations that allow $expectation_A$ to be efficient and semantically transparent. An implementation consistent with this framework must in addition ensure that a horizontal evaluation returns a unique set of samples so that the evaluation of $expectation_A f M$ is deterministic. But even if we are not particularly interested in a sound implementation of $expectation_A$, the horizontal framework has other advantages regarding efficiency in practical applications.

5.2 Horizontal operational semantics

The horizontal operational semantics employs a horizontal evaluation judgment of the form $E \rightsquigarrow Z$ where Z is a *horizontal value*:

$$\text{horizontal value } Z ::= \{(s_1, V_1), \dots, (s_n, V_n)\}$$

The intuition behind $E \rightsquigarrow Z$ is that for each pair $(s, V) \in Z$, we have $E \xrightarrow{s} V$, *i.e.* the expression E evaluates to a value V by consuming a sampling sequence s under the vertical operational semantics. Note that the samples s_i will not actually be carried at runtime—they are used here only to prove the connection with the vertical semantics.

Figure 5 shows the horizontal evaluation rules. $s :: Z$ prepends s to each sampling sequence in Z . In the rule `hee_sample`, we evaluate the expression E with each value V_i from the expression F and then combine resultant horizontal values. In the rule `hee_term`, we make zero or more

$$\begin{array}{c}
\frac{}{\text{efix } \mathbf{x} \dot{\div} A. E \rightsquigarrow \{\}} \text{hee_efix0} \quad \frac{}{Z^D \rightsquigarrow Z} \text{hee_val} \\
\frac{\mathcal{D}_n :: \text{efix } \mathbf{x} \dot{\div} A. E \rightsquigarrow Z_n \quad [Z_n^{D_n}/\mathbf{x}]E \rightsquigarrow Z_{n+1}}{\text{efix } \mathbf{x} \dot{\div} A. E \rightsquigarrow Z_{n+1}} \text{hee_efix}
\end{array}$$

Figure 6: Horizontal evaluation rules for recursive expressions

copies of the value V . In the rule `hee_U`, we generate zero or more random numbers from $U(0.0, 1.0]$. Note that the rules `hee_term` and `hee_U` can return an empty horizontal value so that the evaluation of a recursive expression can terminate.

We can formalize the intuition behind $E \rightsquigarrow Z$ as follows:

THEOREM 5.1. *If $E \rightsquigarrow Z$ and $(s, V) \in Z$, then $E \xrightarrow{s} V$.*

PROOF. By induction over the structure of $E \rightsquigarrow Z$. \square

Theorem 5.1 states that the horizontal operational semantics respects the vertical operational semantics: a horizontal evaluation returns only those samples that can be generated under the vertical operational semantics. In conjunction with Theorem 3.2, it also implies that the horizontal operational semantics satisfies the type preservation property: if $E \rightsquigarrow Z$ and $\cdot \vdash E \dot{\div} A$, then $\cdot \vdash V_i : A$ for any $(s_i, V_i) \in Z$.

Note, however, that the horizontal operational semantics is non-deterministic, and not just probabilistic. It therefore does not directly solve the problem of computing expectations in a semantically sound way. Instead it defines a space of possible implementations, two of which we consider in Section 6.

5.3 Horizontal operational semantics with the expression fixed point construct

Since an expression fixed point construct is a derived form of expression, it does not require a strict extension to the horizontal operational semantics. Nevertheless we develop horizontal evaluation rules for recursive expressions in order to achieve an efficient implementation of the horizontal operational semantics.

We include a horizontal value as a new kind of expression so that we can substitute horizontal values for expression variables:

$$\text{expression } E ::= \dots \mid \mathbf{x} \mid \text{efix } \mathbf{x} \dot{\div} A. E \mid Z^D$$

As an expression, a horizontal value Z is annotated with a horizontal evaluation derivation $\mathcal{D} :: \text{efix } \mathbf{x} \dot{\div} A. E \rightsquigarrow Z$. Thus, from an annotated horizontal value Z^D , we can recover a recursive expression from which we obtain Z through a horizontal evaluation.²

A substitution does not affect annotated horizontal values:

$$\begin{array}{l}
[N/\mathbf{x}]Z^D = Z^D \\
[F/\mathbf{x}]Z^D = Z^D
\end{array}$$

Figure 6 shows the horizontal evaluation rules for recursive expressions. A horizontal evaluation of $\text{efix } \mathbf{x} \dot{\div} A. E$ begins with an empty horizontal value $Z_0 = \{\}$. By substituting Z_n for \mathbf{x} in E and evaluating the resultant expression, we obtain another horizontal value Z_{n+1} for $\text{efix } \mathbf{x} \dot{\div} A. E$.

²We annotate horizontal values only to prove Theorem 5.2. In an actual implementation this is unnecessary.

Now we want to show that the new horizontal evaluation rules also respect the vertical operational semantics as in Theorem 5.1. A simple approach is to prove that the new horizontal evaluation rules are derivable via the translation function $(\cdot)^*$ in Section 3.6. Another approach is to prove that the new horizontal evaluation rules respect the vertical operational semantics augmented with the rule `ee_efix` in Section 3.6. We choose the second because the proof is simpler than in the first.

We define a translation function $(\cdot)^\natural$ such that:

$$(Z^{\text{efix } \mathbf{x} \dot{\div} A. E \rightsquigarrow Z})^\natural = \text{efix } \mathbf{x} \dot{\div} A. E^\natural$$

It enables us to recover a recursive expression from an annotated horizontal value. The translation of all the other forms of terms and expressions is structural.

THEOREM 5.2. *If $E \rightsquigarrow Z$ and $(s, W) \in Z$, then $E^\natural \xrightarrow{s} W^\natural$.*

PROOF. See Appendix. \square

In conjunction with Theorem 3.3, Theorem 5.2 means that the horizontal operational semantics with the new horizontal evaluation rules also respects the vertical operational semantics modulo $(\cdot)^\natural$.

The rule `hee_efix` can be applied to a recursive expression indefinitely, in which case the horizontal evaluation does not terminate. Therefore a recursion depth must be specified for every recursive expression either by programmers or by the runtime system. If we expanded a recursive expression with the translation function $(\cdot)^*$, it would be difficult to specify a recursion depth for it because the rule `te_mfix` does not record a recursion depth.

5.4 Implementation issues

The horizontal operational semantics is a minimal specification that cannot be transformed into an implementation directly: it does not specify how an expression evaluates to a unique horizontal value. For instance, it does not specify how to determine the size of horizontal values. Hence we can consider various implementation strategies ensuring that the result of a horizontal evaluation is unique.

Such an implementation, however, is not sufficient in practice unless it also ensures that an expression E evaluates to a horizontal value that reflects, whether accurately or approximately, the probability distribution denoted by E . As an example, consider a naive implementation in which the rule `hee_term` returns a singleton horizontal value and the rule `hee_U` generates \mathcal{N} random numbers. The following expression denotes a Bernoulli distribution over `True` and `False`, each with a probability 0.5:

```

sample  $x \triangleleft \text{prob } \mathcal{U}$  in
eif  $x \leq 0.5$  then True
else sample dummy  $\triangleleft \text{prob } \mathcal{U}$  in False

```

Note that its evaluation returns a horizontal value containing $\frac{\mathcal{N}}{2}$ `True`'s and $\frac{\mathcal{N}}{2}$ `False`'s in an average case. Even if \mathcal{N} approaches infinity, the horizontal value does not reflect the target probability distribution. Consequently the implementation is not useful in practice because it does not allow us to implement the expectation query.

Therefore a practical implementation of the horizontal operational semantics must be designed in such a way that an

$$\begin{array}{c}
c \otimes Z = \{(V_i, c * c_i) \mid (V_i, c_i) \in Z\} \\
\\
\frac{M \hookrightarrow \text{prob } F \quad F \rightsquigarrow \{(V_i, c_i) \mid i\} \quad [V_i/x]E \rightsquigarrow Z_i \quad |Z_i| = 0 \text{ or } |Z_i| \text{ all equal}}{\text{sample } x \triangleleft M \text{ in } E \rightsquigarrow \bigcup_i c_i \otimes Z_i} \text{hee_sample}_c \\
\\
\frac{M \hookrightarrow V}{M \rightsquigarrow \{(V, c)\}} \text{hee_term}_c \quad \frac{u_i = \frac{i+0.5}{|\mathcal{U}|} \quad i = 0, \dots, |\mathcal{U}| - 1}{\mathcal{U} \rightsquigarrow \{(u_i, c) \mid i\}} \text{hee_U}_c
\end{array}$$

Figure 7: Horizontal evaluation rules with sample counts

expression evaluate to a *unique* horizontal value that reflects, *in a certain way*, the target probability distribution. In the next section, we present two such implementations.

6. IMPLEMENTATION

6.1 Implementation based on sample counts

In the first implementation, we omit sampling sequences from horizontal values because they serve no purpose for the runtime system, and associate each sample V in a horizontal value with a positive integer c called a *sample count*:

$$\text{horizontal value } Z ::= \{(V_1, c_1), \dots, (V_n, c_n)\}$$

A sample count indicates the number of times the corresponding sample is produced. Therefore we can implement the function expectation_A as follows:

$$\frac{M \hookrightarrow \text{prob } E \quad E \rightsquigarrow \{(V_i, c_i) \mid i\} \quad S = \sum_i c_i}{\text{expectation}_A f M \hookrightarrow \sum_i \frac{c_i}{S} f(V_i)} \text{te_expectation}_c$$

The revised horizontal evaluation rules are in Figure 7. Those rules related to recursive expressions are the same as in Figure 6. $c \otimes Z$ multiplies each sample count in Z by c . The length of a horizontal value is defined as the sum of all sample counts in it: $|\{(V_i, c_i) \mid i\}| = \sum_i c_i$. The rule hee_sample_c says that every sample V_i from the expression F contributes equally to the final horizontal value unless $|Z_i| = 0$. If $|Z_i| = 0$, then V_i is ignored. The constant c in the rule hee_term_c or hee_U_c is determined dynamically by constraints imposed by instances of the rule hee_sample_c . The rule hee_U_c generates $|\mathcal{U}|$ random numbers that are equally spaced in $(0.0, 1.0]$, and therefore the constant $|\mathcal{U}|$ determines the precision at which we discretize $U(0.0, 1.0]$. Note that we do not need a random number generator in implementing the rule hee_U_c . It implies that an expression evaluates to a unique horizontal value and that the evaluation of $\text{expectation}_A f M$ is deterministic.

All the rules above conform to the intuition that an expression evaluates to a horizontal value that best approximates the target probability distribution with a limited number of samples. We can formalize this intuition by showing that as $|\mathcal{U}|$ approaches infinity, the horizontal value also approaches an accurate representation of the target probability distribution. For a recursive expression, we need an additional assumption that the recursion depth also approaches infinity.

As an example, consider the evaluation of *geometric_fix p*. For the sake of simplicity, we assume $p = \frac{n}{m}$ and $|\mathcal{U}| = mk$

$$\begin{array}{c}
w \odot Z = \{(V_i, w * w_i) \mid (V_i, w_i) \in Z\} \\
\text{normalize } \{\} = \{\} \\
\text{normalize } \{(V_i, w_i) \mid i\} = \{(V_i, \frac{w_i}{S}) \mid i\} \text{ where } S = \sum_i w_i \\
\\
\frac{M \hookrightarrow \text{prob } F \quad F \rightsquigarrow \{(V_i, w_i) \mid i\} \quad [V_i/x]E \rightsquigarrow Z_i}{\text{sample } x \triangleleft M \text{ in } E \rightsquigarrow \text{normalize } \bigcup_i w_i \odot Z_i} \text{hee_sample}_w \\
\\
\frac{M \hookrightarrow V}{M \rightsquigarrow \{(V, 1.0)\}} \text{hee_term}_w \quad \frac{u_i = \frac{i+0.5}{|\mathcal{U}|} \quad i = 0, \dots, |\mathcal{U}| - 1}{\mathcal{U} \rightsquigarrow \{(u_i, \frac{1.0}{|\mathcal{U}|}) \mid i\}} \text{hee_U}_w
\end{array}$$

Figure 8: Horizontal evaluation rules with sample weights

where m and n are positive integers. We can show that with a recursion depth $d \geq 1$, the evaluation returns a horizontal value Z_d such that $|Z_d| = m^{d-1}nk^d$, $\sum_{(i,c) \in Z_d} c = m^{d-i-2}n^2(m-n)^i k^d$ where $i < d-1$, and $\sum_{(d-1,c) \in Z_d} c = n(m-n)^{d-1}k^d$. Therefore, as both k and d approach infinity, the proportion of a sample i in Z_d becomes $\frac{m^{d-i-2}n^2(m-n)^i k^d}{m^{d-1}nk^d} = p(1.0-p)^i$, which is exactly the probability for the sample i in a geometric distribution with parameter p .

Although the above implementation is an improvement over the naive implementation of the horizontal operational semantics, it has two practical problems. First we cannot easily encode probability distributions beyond the precision determined by the constant $|\mathcal{U}|$. For instance, with $|\mathcal{U}| = 10$, *bernoulli 0.1* and *bernoulli 0.11* are indistinguishable by the runtime system while they denote different probability distributions. Second the evaluation of a recursive expression may suffer from exponential growth in the size of horizontal values, preventing us from choosing a large recursion depth. Even if we compress a horizontal value by combining those entries containing the same sample, sample counts may still grow exponentially, producing overflows in integer arithmetic. In the next subsection, we present another implementation that alleviates these problems.

6.2 Implementation based on sample weights

In the second implementation, we employ the idea from the sample-based representation of probability distributions [7] and associate each sample V in a horizontal value with a positive real number w called a *sample weight*:

$$\text{horizontal value } Z ::= \{(V_1, w_1), \dots, (V_n, w_n)\} \text{ where } n = 0 \text{ or } \sum_i w_i = 1.0$$

In terms of the previous implementation, we can think of a sample weight as a sample count divided by the sum of all sample counts in a given horizontal value. Since the sum of all sample weights in a non-empty horizontal value is 1.0, we can implement the function expectation_A as follows:

$$\frac{M \hookrightarrow \text{prob } E \quad E \rightsquigarrow \{(V_i, w_i) \mid i\}}{\text{expectation}_A f M \hookrightarrow \sum_i w_i f(V_i)} \text{te_expectation}_w$$

The revised horizontal evaluation rules are in Figure 8. Those rules related to recursive expressions are the same as in Figure 6. $w \odot Z$ multiplies each sample weight in Z by w . The function normalize takes a set of weighted samples as input and returns a horizontal value by normalizing it. We need to apply normalize in the rule hee_sample_w because

Z_i may be empty. As in the previous implementation, an expression evaluates to a unique horizontal value, and the evaluation of $expectation_A f M$ is deterministic.

The use of sample weights in horizontal values enables us to introduce as a primitive construct a *binary choice expression* $choose_p M_1 M_2$ whose horizontal evaluation rule is:

$$\frac{M_1 \hookrightarrow V_1 \quad M_2 \hookrightarrow V_2}{choose_p M_1 M_2 \rightsquigarrow \{(V_1, p), (V_2, 1.0 - p)\}} \text{hee_choose}_c$$

Note that a binary choice expression is not a primitive construct under the vertical operational semantics because we can simulate it in the same way as we encode Bernoulli distributions. By composing multiple binary choice expressions, we can now represent accurately any probability distribution over a finite domain.³ As a simple example, we can encode a Bernoulli distribution over type `bool` as follows:

```
let bernoulli = λp:real. prob choose_p True False
```

The binary choice expression eliminates the first problem with the previous implementation at least for discrete probability distributions. The problem still remains for continuous probability distributions, but this is acceptable considering the nature of continuous probability distributions: we cannot represent an arbitrary continuous probability distribution accurately with a finite number of samples. We resolve the second problem as follows: whenever we obtain a horizontal value, we discard those samples whose sample weights are below a threshold \mathcal{E} , and then normalize the resultant set. We also try to compress horizontal values when we evaluate recursive expressions. By choosing a small value for \mathcal{E} , we can achieve high precision in approximating probability distributions, which is demonstrated in the next section.

7. EXPERIMENTAL RESULTS

We implement the vertical operational semantics and the horizontal operational semantics with sample weights, and test them on seven probability distributions from Section 4 with the expectation query. Both implementations are written in Objective Caml 3.06, and all experiments are performed on Pentium III 500Mhz with 384 MBytes memory.

Figure 9 shows the experimental results for the vertical operational semantics together with true means and variances.⁴ For each test case, we generate 1000000 samples to compute its mean and variance with the expectation query. Since a uniform expression evaluates to a random number generated from $U(0.0, 1.0)$, the results are different for each run on the test cases; the results in Figure 9 are obtained from a particular run on the test cases.

Figure 10 shows the experimental results for the horizontal operational semantics with sample weights where we use a threshold $\mathcal{E} = 1.0^{-10}$. We reimplement Bernoulli distributions with binary choice expressions as explained in Section 6.2. For each test case, we set $|U|$ if it contains uniform expressions, set the recursion depth if it is a recursive expression, and compute its mean and variance with the expectation query. The number of samples in the final horizontal value is not readily predictable if it is a recursive expression.

³In fact, $choose_{0.5} M_1 M_2$ suffices as long as the probability for each element in the domain is computable [9].

⁴For the parameters used in $Q_{burglary|John_calls}$, see [31]. Its mean refers to $p_{burglary|John_calls}$ in Section 4.

Since a uniform expression evaluates to a unique horizontal value, the results are always the same.

The results from *gaussian_rejection* and $Q_{burglary|John_calls}$ suggest that when we evaluate a recursive expression, the number of samples and the execution time may grow exponentially with $|U|$ or the recursion depth. For recursive expressions denoting discrete probability distributions, the number of samples does not grow indefinitely: once the horizontal value approaches an accurate representation of the target probability distribution, the number of samples remains stable. For instance, even if we increase the recursion depth from 200 to 2000, $Q_{burglary|John_calls}$ returns a horizontal value containing only 46708 samples, *i.e.* 2317 more samples (in 280.17 seconds).

We observe that the horizontal operational semantics with sample weights consistently outperforms the vertical operational semantics with respect to accuracy in mean and variance. The only exception is the variance in *gaussian_rejection*, but still a precise mean compensates for the increase in variance. The reduction in execution time is also significant, especially in *binomial*, *geometric_fix*, and $Q_{burglary|John_calls}$. It is mainly due to the use of binary choice expressions in implementing Bernoulli distributions. The result from $Q_{burglary|John_calls}$ is remarkable because such an accurate mean is hard to obtain with the vertical operational semantics.

8. RELATED WORK

In this section, we discuss previous work on probabilistic languages based upon monads or sampling functions. For other probabilistic languages, see [32, 19, 11, 34, 25, 27].

Ramsey and Pfeffer [29] present a stochastic lambda calculus whose denotational semantics is based upon the monad of probability measures. A lambda abstraction translates into a function from values to probability distributions, and corresponds to a sampling expression in our language. A binary choice construct $choose\ p\ e_1\ e_2$ corresponds to a uniform expression in our language in the sense that it enables us to create probability distributions other than point mass distributions. The paper presents a sampling monad, but it plays a different role from what the sampling monad \circ does in our language: their sampling monad is a means of implementing the sampling query whereas the sampling monad \circ in our language serves as its semantic foundation.

Jones [18, 17] presents a probabilistic metalanguage whose denotational semantics is based upon the monad of evaluations. The language syntactically distinguishes expressions denoting probability distributions from *function expressions*, which denote functions from values to probability distributions. Thus a function expression corresponds to a sampling expression in our language. The language also provides a binary choice construct $e_1\ \text{or}_p\ e_2$.

Kozen [20] investigates the semantics of probabilistic while programs. A random assignment $x := \text{random}$ assigns a random number to a variable x and is the source of probability distributions. It can be thought of as a primitive construct for sampling functions because the variable x can be used later as an argument to a sampling function. The language draws no distinction between discrete and continuous probability distributions, which is a property of a probabilistic language whose mathematical basis is sampling functions.

The first author [24] presents a probabilistic calculus λ_γ whose mathematical basis is sampling functions. In order to

probability distributions	true mean	true variance	time (seconds)	mean	variance
<i>uniform</i> 0.0 1.0	0.5	0.083333	0.93	0.500077	0.083398
<i>binomial</i> 0.5 50	25.0	12.5	72.42	25.000047	12.501351
<i>binomial</i> 0.5 100	50.0	25.0	139.15	50.005734	25.013649
<i>geometric_fix</i> 0.5	1.0	2.0	7.58	1.000163	2.002239
<i>gaussian_rejection</i> 0.0 1.0	0.0	1.0	5.76	0.001421	1.000043
<i>gaussian_BoxMuller</i> 0.0 1.0	0.0	1.0	3.11	0.000543	0.998157
<i>gaussian_central</i> 0.0 1.0	0.0	1.0	3.15	-0.000708	0.997768
$Q_{\text{burglary} \text{John_calls}}$	0.016444	-	270.42	0.016599	-

Figure 9: Experimental results for the vertical operational semantics

probability distributions	$ \mathcal{U} $	recursion depth	# of samples	time (seconds)	mean	variance
<i>uniform</i> 0.0 1.0	1000000	-	1000000	0.33	0.500000	0.083333
<i>binomial</i> 0.5 50	-	-	100	0.02	25.000000	12.500000
<i>binomial</i> 0.5 100	-	-	200	0.14	50.000000	25.000000
<i>geometric_fix</i> 0.5	-	10	10	0.00	0.998047	1.962887
	-	100	33	0.01	1.000000	2.000000
<i>gaussian_rejection</i> 0.0 1.0	8	5	24880	0.03	0.000000	1.064119
	16	5	928672	2.87	0.000000	1.004040
<i>gaussian_BoxMuller</i> 0.0 1.0	1000	-	1000000	2.02	0.000000	0.999653
<i>gaussian_central</i> 0.0 1.0	100	-	1000000	1.47	0.000000	0.999900
$Q_{\text{burglary} \text{John_calls}}$	-	20	1560	0.02	0.010861	-
	-	200	44391	15.09	0.016443	-

Figure 10: Experimental results for the horizontal operational semantics with sample weights

encode sampling functions directly, the language provides a *sampling construct* $\gamma.e$ where γ is a formal parameter and e is the body of a sampling function. The evaluation of $\gamma.e$ proceeds by generating a random number from $U(0.0, 1.0]$ and substituting it for γ in e , and hence the operational semantics is vertical. The idea of using sampling functions as the mathematical basis in our language is adopted from this paper.

9. CONCLUSION

We have presented a monadic probabilistic language whose mathematical basis is sampling functions. Our language is capable of specifying probability distributions over infinite discrete domains and continuous domains, and still employs a unified representation scheme for probability distributions. This is a property not found in previous work on monadic probabilistic languages. We have also presented an efficient implementation of the expectation query using a horizontal operational semantics. We believe that the idea of the horizontal operational semantics can be applied to any probabilistic language employing a vertical operational semantics.

One limitation of our work is that in the horizontal operational semantics, there are no optimal values for the constant $|\mathcal{U}|$ and the recursion depth universally applicable to all expressions. Such values must be specified either by programmers or inferred by the runtime system for each target expression.

We currently have a prototype implementation of our language embedded in Objective Caml. In prior experiments we have explored using functional implementations of realistic algorithms for robot localization and mapping. Our most immediate objective is to recast these implementations

entirely in the monadic language and carry out further experimental analyses.

Acknowledgment

The authors are grateful to Jonathan Moody and Normal Ramsey for their helpful discussion on probability monads.

10. REFERENCES

- [1] P. N. Benton, G. M. Bierman, and V. C. V. de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, Mar. 1998.
- [2] P. Bratley, B. Fox, and L. Schrage. *A guide to simulation*. Springer Verlag, 2nd edition, 1996.
- [3] E. Charniak. *Statistical Language Learning*. MIT Press, Cambridge, Massachusetts, 1993.
- [4] F. Dellaert, S. Seitz, C. Thorpe, and S. Thrun. EM, MCMC, and chain flipping for structure from motion with unknown correspondence. *Machine Learning*, 50(1-2):45–71, 2003.
- [5] M. Fairtlough and M. Mendler. Propositional lax logic. *Information and Computation*, 137(1):1–33, Aug. 1997.
- [6] A. Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.
- [7] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *AAAI-99*, pages 343–349. AAAI/MIT Press, 1999.
- [8] W. Gilks, S. Richardson, and D. Spiegelhalter, editors. *Markov Chain Monte Carlo in Practice*. Chapman and Hall/CRC, 1996.

- [9] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.
- [10] M. Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes In Mathematics*, pages 68–85. Springer Verlag, 1981.
- [11] V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *26th ACM POPL*, pages 189–202. ACM Press, 1999.
- [12] B. Hannaford and P. Lee. Hidden markov model analysis of force torque information in telemanipulation. *International Journal of Robotics Research*, 10(5):528–539, 1991.
- [13] R. Harper and F. Pfenning. Operational semantics for a modal type system of effects. Preprint, 2003.
- [14] J. Henderson, S. Salzberg, and K. Fasman. Finding genes in human DNA with a hidden Markov model. In *Proceedings of Intelligent Systems for Molecular Biology '96*, Washington University, St. Louis, MI, 1996.
- [15] M. Isard and A. Blake. CONDENSATION: conditional density propagation for visual tracking. *International Journal of Computer Vision*, 29(1):5–28, 1998.
- [16] F. Jelinek. *Statistical Methods for Speech Recognition (Language, Speech, and Communication)*. MIT Press, Boston, MA, 1998.
- [17] C. Jones. *Probabilistic Non-Determinism*. PhD thesis, Department of Computer Science, University of Edinburgh, 1990.
- [18] C. Jones and G. D. Plotkin. A probabilistic power-domain of evaluations. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 186–195. IEEE Computer Society Press, June 1989.
- [19] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *AAAI-97/IAAI-97*, pages 740–747. AAAI Press, 1997.
- [20] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.
- [21] A. Krogh, I. Mian, and D. Haussler. A hidden markov model that finds genes in *E. coli* DNA. *Nucleic Acids Research*, 22:4768–4778, 1994.
- [22] E. Moggi. Computational lambda-calculus and monads. In *LICS-89*, pages 14–23. IEEE Computer Society Press, 1989.
- [23] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
- [24] S. Park. A calculus for probabilistic languages. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 38–49. ACM Press, 2003.
- [25] A. Pfeffer. IBAL: A probabilistic rational programming language. In *IJCAI-01*, pages 733–740. Morgan Kaufmann Publishers, 2001.
- [26] F. Pfenning and R. Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.
- [27] D. Pless and G. Luger. Toward general analysis of recursive probability models. In *UAI-01*, pages 429–436. Morgan Kaufmann Publishers, 2001.
- [28] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*. IEEE, 1989. IEEE Log Number 8825949.
- [29] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *29th ACM POPL*, pages 154–165. ACM Press, 2002.
- [30] W. Rudin. *Real and Complex Analysis*. McGraw-Hill, New York, 3 edition, 1986.
- [31] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
- [32] N. Saheb-Djahromi. Probabilistic LCF. In *Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science*, volume 64 of *LNCS*, pages 442–451. Springer, 1978.
- [33] R. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainty. *International Journal of Robotics Research*, 5(4):56–68, 1986.
- [34] S. Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *ICRA-00*. IEEE, 2000.
- [35] S. Thrun. Is robotics going statistics? The field of probabilistic robotics. *Communications of the ACM*, March 2001.
- [36] A. Waibel and K.-F. Lee, editors. *Readings in Speech Recognition*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

Appendix

Proof of Theorem 3.3

PROPOSITION 10.1. For any term N , we have $([N/x]M)^* = [N^*/x]M^*$ and $([N/x]E)^* = [N^*/x]E^*$. For any expression F , we have $([F/x]M)^* = [F^*/x]M^*$ and $([F/x]E)^* = [F^*/x]E^*$.

PROOF. By simultaneous induction on the structure of M and E . \square

PROPOSITION 10.2. $(\text{efix } \mathbf{x} \div A. E)^* \xrightarrow{s} V$ if and only if $([\text{efix } \mathbf{x} \div A. E]^*/\mathbf{x})E^* \xrightarrow{s} V$.

PROOF. We observe that $(\text{efix } \mathbf{x} \div A. E)^* \xrightarrow{s} V$ expands to

$$\frac{\dots \quad [(\text{efix } \mathbf{x} \div A. E)^*/\mathbf{x}]E^* \xrightarrow{s} V \quad [V/x_r]x_r \xrightarrow{c} V}{(\text{efix } \mathbf{x} \div A. E)^* \xrightarrow{s} V} \quad \square$$

LEMMA 10.3. If $M \hookrightarrow V$, then $M^* \hookrightarrow V^*$. If $E \xrightarrow{s} W$ with the rule `ee_efix`, then $E^* \xrightarrow{s} W^*$.

PROOF. By simultaneous induction over the structure of $M \hookrightarrow V$ and $E \xrightarrow{s} W$. For the case of $E = \text{efix } \mathbf{x} \div A. F$, we use Proposition 10.2. \square

LEMMA 10.4. If $M^* \hookrightarrow V'$, then $M \hookrightarrow V$ and $V' = V^*$. If $E^* \xrightarrow{s} W'$, then $E \xrightarrow{s} W$ with the rule `ee_efix` and $W' = W^*$.

PROOF. By simultaneous induction over the structure of $M^* \hookrightarrow V'$ and $E^* \xrightarrow{s} W'$. For the case of $E = \text{efix } \mathbf{x} \div A. F$, we use the observation in the proof of Proposition 10.2. \square

Theorem 3.3 follows from Lemmas 10.3 and 10.4.

Proof of Theorem 5.2

PROPOSITION 10.5. For any term N , we have $([N/x]M)^\natural = [N^\natural/x]M^\natural$ and $([N/x]E)^\natural = [N^\natural/x]E^\natural$.

PROOF. By simultaneous induction on the structure of M and E . For the case of $E = Z^{\text{efix } \mathbf{y} \div A. F \rightsquigarrow Z}$, we use the fact that $\text{efix } \mathbf{y} \div A. F^\natural$ is closed. \square

PROPOSITION 10.6. *For any expression F , we have $([F/\mathbf{x}]M)^\natural = [F^\natural/\mathbf{x}]M^\natural$ and $([F/\mathbf{x}]E)^\natural = [F^\natural/\mathbf{x}]E^\natural$.*

PROOF. By simultaneous induction on the structure of M and E . For the case of $E = Z^{\text{efix } \mathbf{y} \div A. F \rightsquigarrow Z}$, we use the fact that $\text{efix } \mathbf{y} \div A. F^\natural$ is closed. \square

Suppose that we have derivations $M \hookrightarrow V$ and $E \rightsquigarrow Z$ where $(s, W) \in Z$. We define $[M \hookrightarrow V]^\mathcal{M}$ and $[E \rightsquigarrow Z]_{s, W}^\mathcal{E}$ inductively as in Figure 11.

LEMMA 10.7. *If $M \hookrightarrow V$, then $[M \hookrightarrow V]^\mathcal{M}$ is a derivation of $M^\natural \hookrightarrow V^\natural$. If $E \rightsquigarrow Z$ and $(s, W) \in Z$, then $[E \rightsquigarrow Z]_{s, W}^\mathcal{E}$ is a derivation of $E^\natural \xrightarrow{s} W^\natural$.*

PROOF. By simultaneous induction over the structure of $M \hookrightarrow V$ and $E \rightsquigarrow Z$. \square

Theorem 5.2 follows from Lemma 10.7.

$$\begin{aligned}
& \left[\frac{\overline{\lambda x : A. M \hookrightarrow \lambda x : A. M}^{\mathcal{M}}}{\frac{M_1 \hookrightarrow \lambda x : A. M \quad [M_2/x]M \hookrightarrow V}{M_1 M_2 \hookrightarrow V}} \right]^{\mathcal{M}} = \frac{\overline{\lambda x : A. M^{\natural} \hookrightarrow \lambda x : A. M^{\natural}}}{\frac{[M_1 \hookrightarrow \lambda x : A. M]^{\mathcal{M}} \quad [[M_2/x]M \hookrightarrow V]^{\mathcal{M}}}{M_1^{\natural} M_2^{\natural} \hookrightarrow V^{\natural}}} \\
& \left[\frac{\overline{\text{prob } E \hookrightarrow \text{prob } E}^{\mathcal{M}}}{\overline{u \hookrightarrow u}^{\mathcal{M}}} \right]^{\mathcal{M}} = \frac{\overline{\text{prob } E^{\natural} \hookrightarrow \text{prob } E^{\natural}}}{\overline{u \hookrightarrow u}} \\
& \left[\frac{\overline{[\text{fix } x : A. M/x]M \hookrightarrow V}^{\mathcal{M}}}{\text{fix } x : A. M \hookrightarrow V} \right]^{\mathcal{M}} = \frac{[[\text{fix } x : A. M/x]M \hookrightarrow V]^{\mathcal{M}}}{\text{fix } x : A. M^{\natural} \hookrightarrow V^{\natural}} \\
& \left[\frac{M \hookrightarrow \text{prob } F \quad F \rightsquigarrow \{(s_i, W_i) \mid i\} \quad [W_i/x]E \rightsquigarrow Z_i}{\text{sample } x \triangleleft M \text{ in } E \rightsquigarrow \bigcup_i s_i :: Z_i} \right]_{s_i, s', W}^{\mathcal{E}} = \frac{[M \hookrightarrow \text{prob } F]^{\mathcal{M}} \quad [F \rightsquigarrow \{(s_i, W_i) \mid i\}]_{s_i, W_i}^{\mathcal{E}} \quad [[W_i/x]E \rightsquigarrow Z_i]_{s', W}^{\mathcal{E}}}{\text{sample } x \triangleleft M^{\natural} \text{ in } E^{\natural} \xrightarrow{s_i, s'} W^{\natural}} \\
& \left[\frac{M \hookrightarrow W}{M \rightsquigarrow \{(\epsilon, W), \dots, (\epsilon, W)\}} \right]_{\epsilon, W}^{\mathcal{E}} = \frac{[M \hookrightarrow W]^{\mathcal{M}}}{M^{\natural} \xrightarrow{\epsilon} W^{\natural}} \\
& \left[\frac{u_i \sim U(0.0, 1.0)}{\mathcal{U} \rightsquigarrow \{(u_i \epsilon, u) \mid i\}} \right]_{u_i \epsilon, u_i}^{\mathcal{E}} = \frac{u_i \sim U(0.0, 1.0)}{\mathcal{U} \xrightarrow{u_i \epsilon} u_i} \\
& \left[\frac{\mathcal{D}_n :: \text{efix } \mathbf{x} \div A. E \rightsquigarrow Z_n \quad [Z_n^{\mathcal{D}_n}/\mathbf{x}]E \rightsquigarrow Z_{n+1}}{\text{efix } \mathbf{x} \div A. E \rightsquigarrow Z_{n+1}} \right]_{s, W}^{\mathcal{E}} = \frac{[[Z_n^{\mathcal{D}_n}/\mathbf{x}]E \rightsquigarrow Z_{n+1}]_{s, W}^{\mathcal{E}}}{\text{efix } \mathbf{x} \div A. E^{\natural} \xrightarrow{s} W^{\natural}} \\
& \left[\overline{Z^{\mathcal{D}} \rightsquigarrow Z} \right]_{s, W}^{\mathcal{E}} = [\mathcal{D}]_{s, W}^{\mathcal{E}}
\end{aligned}$$

Figure 11: Translation of $M \hookrightarrow V$ and $E \rightsquigarrow Z$