A Probabilistic Language based upon Sampling Functions

Sungwoo Park
Computer Science and Engineering Department
Pohang University of Science and Technology
gla@postech.ac.kr

Frank Pfenning
Computer Science Department
Carnegie Mellon University
fp@cs.cmu.edu

Sebastian Thrun
Computer Science Department
Stanford University
thrun@stanford.edu

As probabilistic computations play an increasing role in solving various problems, researchers have designed probabilistic languages which treat probability distributions as primitive datatypes. Most probabilistic languages, however, focus only on discrete distributions and have limited expressive power. This paper presents a probabilistic language, called $\lambda_{\bigcirc}$, whose expressive power is beyond discrete distributions. Rich expressiveness of $\lambda_{\bigcirc}$ is due to its use of *sampling functions*, *i.e.*, mappings from the unit interval $(0.0, 1.0]$ to probability domains, in specifying probability distributions. As such, $\lambda_{\bigcirc}$ enables programmers to formally express and reason about sampling methods developed in simulation theory. The use of $\lambda_{\bigcirc}$ is demonstrated with three applications in robotics: robot localization, people tracking, and robotic mapping. All experiments have been carried out with real robots.

Categories and Subject Descriptors: D.3.2 [**Language Classifications**]: Specialized application languages

General Terms: Languages, Experimentation

Additional Key Words and Phrases: Probabilistic language, Probability distribution, Sampling function, Robotics

1. INTRODUCTION

A probabilistic computation is a computation that makes probabilistic choices or whose result is a probability distribution. As an alternative paradigm to deterministic computation, it has been used successfully in diverse fields of computer science such as speech recognition [Rabiner 1989; Jelinek 1998], natural language processing [Charniak 1993], computer vision [Isard and Blake 1998], randomized algorithms [Motwani and Raghavan 1995], and robotics [Thrun 2000a]. Its success

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2008 ACM 0164-0925/2008/0500-0001 $5.00

lies in the fact that probabilistic approaches often overcome the practical limitation of deterministic approaches. A trivial example is the problem of testing whether a multivariate polynomial given by a program without branch statements is identically zero or not. It is difficult to find a practical deterministic solution, but there is a simple probabilistic solution: evaluate the polynomial on a randomly chosen input and check if the result is zero.

As probabilistic computations play an increasing role in solving various problems, researchers have also designed *probabilistic languages* to facilitate their implementation [Jones 1990; Koller et al. 1997; Gupta et al. 1999; Thrun 2000b; Pfeffer 2001; Ramsey and Pfeffer 2002; Mogensen 2002; Park 2003]. A probabilistic language treats probability distributions as built-in datatypes and thus abstracts from *representation schemes*, *i.e.*, specific data structures for representing probability distributions. As a result, it allows programmers to concentrate on how to formulate probabilistic computations, or probabilistic algorithms, at the level of probability distributions in the sense that no particular representation scheme is assumed. The translation of such a formulation in a probabilistic language (by programmers) produces concise and elegant code implementing the target probabilistic computation, which is typical of a probabilistic language. Then we can classify probabilistic languages according to their expressive power, *i.e.*, the set of probability distributions that can be encoded.

A typical probabilistic language supports at least discrete distributions, for which there exists a representation scheme sufficient for all practical purposes: a set of pairs consisting of a value from the probability domain and its probability. We could use such a probabilistic language for those problems involving only discrete distributions. If non-discrete distributions are involved, however, we usually use a conventional language for the sake of efficiency, assuming a specific kind of probability distributions (*e.g.*, Gaussian distributions) or choosing a specific representation scheme (*e.g.*, a set of samples from the probability distribution). For this reason, there has been little effort to develop probabilistic languages whose expressive power is beyond discrete distributions.

This paper presents a probabilistic language, called $\lambda_{\bigcirc}$, which supports not only discrete distributions but also continuous distributions and even those belonging to neither group. The main contributions of this paper are summarized as follows:

— $\lambda_{\bigcirc}$ uses *sampling functions*, *i.e.*, mappings from the unit interval $(0.0, 1.0]$ to probability domains, to specify probability distributions. Sampling functions serve as a mathematical basis for probabilistic languages whose expressive power is beyond discrete distributions.

— $\lambda_{\bigcirc}$ serves as a programming language in which sampling methods developed in simulation theory [Bratley et al. 1996] can be formally expressed and reasoned about.

— The use of $\lambda_{\bigcirc}$ is demonstrated with three applications in robotics: robot localization, people tracking, and robotic mapping. Thus $\lambda_{\bigcirc}$ serves as an example of high-level language applied to a problem domain where imperative languages have been traditionally dominant.

We now describe the design of $\lambda_{\bigcirc}$ at a conceptual level. Its implementation and applications are also briefly discussed.

Notation If a variable x ranges over the domain of a probability distribution P, then $P(x)$ means, depending on the context, either the probability distribution itself (as in "probability distribution $P(x)$") or the probability of a particular value x (as in "probability $P(x)$"). We write $P(x)$ for probability distribution P when we want to emphasize the use of variable x. If we do not need a specific name for a probability distribution, we use $Prob$ (as in "probability distribution $Prob(x)$"). Similarly $P(x|y)$ means either the conditional probability P itself or the probability of x conditioned on y. We write P_y or $P(\cdot|y)$ for the probability distribution conditioned on y. We write $U(0.0, 1.0]$ for a uniform distribution over the unit interval $(0.0, 1.0]$.

Sampling functions as the mathematical basis

The expressive power of a probabilistic language, *i.e.*, the set of probability distributions expressible in a probabilistic language, is determined to a large extent by its mathematical basis. In the case of $\lambda_{\bigcirc}$, we intend to support all kinds of probability distributions without drawing a syntactic or semantic distinction so as to achieve a uniform framework for probabilistic computation. Consequently the mathematical basis of $\lambda_{\bigcirc}$ cannot be what is applicable only to a specific kind of probability distributions. Examples are probability mass functions which are specific to discrete distributions, probability density functions which are specific to continuous distributions, and cumulative distribution functions which assume an ordering on each probability domain.

Probability measures [Rudin 1986] are a possibility because they are synonymous with probability distributions. A probability measure μ over a domain $\mathcal{D}$ conceptually maps the set of subsets of $\mathcal{D}$ (or, the set of events on $\mathcal{D}$) to probabilities in $[0.0, 1.0]$. Probability measures are, however, not a practical choice as the mathematical basis because they are difficult to represent in a data structure if the domain is infinite. As an example, consider a continuous probability distribution P of the position of a robot in a two-dimensional environment. (Since P is continuous, the domain is infinite even if the environment is physically finite.) The probability measure μ corresponding to P should be able to calculate a probability for any given part of the environment (as opposed to a particular spot in the environment) — whether it is a contiguous region or a collection of disjoint regions, or whether it rectangular or oval-shaped. Thus finding a suitable data structure for μ involves the problem of representing an arbitrary part of the environment, and is thus far from a routine task.

The main idea behind the design of $\lambda_{\bigcirc}$ is that we can specify a probability distribution indirectly by answering *"How can we generate samples from it?"*, or equivalently, by providing a sampling function for it. A sampling function is defined as a mapping from the unit interval $(0.0, 1.0]$ to a probability domain $\mathcal{D}$. Given a random number drawn from $U(0.0, 1.0]$, it returns a sample in $\mathcal{D}$, and thus specifies a unique probability distribution. We choose sampling functions as the mathematical basis of $\lambda_{\bigcirc}$.

In specifying how to generate samples, we wish to exploit sampling methods developed in simulation theory, most of which consume multiple (independent) random numbers to produce a single sample. To this end, we use a generalized notion of sampling function which maps $(0.0, 1.0]^{\infty}$ to $\mathcal{D} \times (0.0, 1.0]^{\infty}$ where $(0.0, 1.0]^{\infty}$ denotes an infinite product of $(0.0, 1.0]$. Operationally a sampling function now takes as input an infinite sequence of random numbers drawn independently from $U(0.0, 1.0]$, consumes zero or more random numbers, and returns a sample with the remaining sequence. The use of generalized sampling functions as the mathematical basis also allows programmers to transcribe correctness proofs of sampling methods in simulation theory to correctness proofs of encodings in $\lambda_{\bigcirc}$.

Monte Carlo methods in $\lambda_{\bigcirc}$

In $\lambda_{\bigcirc}$, a sampling function is represented by a probabilistic computation that consumes zero or more random numbers (rather than a single random number) drawn from $U(0.0, 1.0]$. In the context of data abstraction, it means that a probability distribution is *constructed* from such a probabilistic computation. The expressive power of $\lambda_{\bigcirc}$ allows programmers to construct (or encode) different kinds of probability distributions in a uniform way. Equally important is the question of how to *observe* (or reason about) a given probability distribution, *i.e.*, how to get information out of it, through various queries.

Since a probabilistic computation in $\lambda_{\bigcirc}$ only describes a procedure for generating samples, the only way to observe a probability distribution is by generating samples from it. As a result, $\lambda_{\bigcirc}$ is limited in its support for queries on probability distributions. For example, it does not permit a precise implementation of such queries as means, variances, and probabilities of specific events. $\lambda_{\bigcirc}$ alleviates this limitation by exploiting the Monte Carlo method [MacKay 1998], which approximately answers a query on a probability distribution by generating a large number of samples and then analyzing them.

Due to the nature of the Monte Carlo method, the cost of answering a query is proportional to the number of samples used in the analysis. The cost of generating a single sample is determined by the specific procedure chosen by programmers, rather than by the probability distribution itself from which to draw samples. For example, a geometric distribution can be encoded either with a recursive procedure that simulates coin tosses until a certain outcome is observed, or by a simple transformation (called the *inverse transform method*) which requires only a single random number. These two methods of encoding the same probability distribution differ in the cost of generating a single sample and hence in the cost of answering the same query by the Monte Carlo method. For a similar reason, the accuracy of the result of the Monte Carlo method, which improves with the number of samples, is also affected by the specific procedure chosen by programmers. As a programming language to encode probability distributions, $\lambda_{\bigcirc}$ itself does not provide a tool to estimate the cost or accuracy associated with the Monte Carlo method.

Measure-theoretic view of sampling functions

While the accepted mathematical basis of probability theory is measure theory [Rudin 1986], we do not investigate measure-theoretic properties of sampling functions definable in $\lambda_{\bigcirc}$. In fact, the presence of fixed point constructs in $\lambda_{\bigcirc}$ (for recursive computations which may consume an arbitrary number of random numbers) seems to make it difficult even to define measurable spaces to which the unit interval is mapped, since fixed point constructs require domain-theoretic structures in order to solve recursive equations.

Every probabilistic computation expressed in $\lambda_{\bigcirc}$ is easily translated into a generalized sampling function (which takes $(0.0, 1.0]^\infty$ as input). We have not investigated if generalized sampling functions definable in $\lambda_\bigcirc$ are all measurable. Nevertheless generalized sampling functions definable in $\lambda_\bigcirc$ are shown to be closely connected with sampling methods from simulation theory, which are widely agreed to be a form of probabilistic computation.

Implementation of $\lambda_\bigcirc$

Instead of implementing $\lambda_\bigcirc$ as a complete programming language of its own, we embed it in an existing functional language by building a translator. Specifically we extend the syntax of Objective CAML[1] to incorporate the syntax of $\lambda_\bigcirc$, and then translate language constructs of $\lambda_\bigcirc$ back into the original syntax. The translator is sound and complete in the sense that both type and reducibility of any program in $\lambda_\bigcirc$, whether well-typed/reducible or ill-typed/irreducible, are preserved when translated in Objective CAML. Normal forms in $\lambda_\bigcirc$ are also preserved thanks to the use of an abstract datatype in the translation.

Applications to robotics

An important part of our work is to demonstrate the use of $\lambda_\bigcirc$ by applying it to real problems. As the main testbed, we choose *robotics* [Thrun 2000a]. It offers a variety of real problems that necessitate probabilistic computations over continuous distributions. We use $\lambda_\bigcirc$ for three applications in robotics: robot localization [Thrun 2000a], people tracking [Montemerlo et al. 2002], and robotic mapping [Thrun 2002]. In each case, the state of a robot is represented by a probability distribution, whose update equation is formulated at the level of probability distributions and translated in $\lambda_\bigcirc$. All experiments in our work have been carried out with real robots.

A comparison between our robot localizer and another written in C gives evidence that the benefit of implementing probabilistic computations in $\lambda_\bigcirc$, such as readability and conciseness of code, can outweigh its disadvantage in speed. Thus $\lambda_\bigcirc$ serves as another example of high-level language whose power is well exploited in a problem domain where imperative languages have been traditionally dominant.

Organization of the paper

This paper is organized as follows. Section 2 presents an example that illustrates the disadvantage of conventional languages in implementing probabilistic computations and also motivates the development of $\lambda_\bigcirc$. Section 3 presents the type system and the operational semantics of $\lambda_\bigcirc$. Section 4 shows how to encode various probability distributions in $\lambda_\bigcirc$ and demonstrates properties of $\lambda_\bigcirc$. Section 5 shows how to prove the correctness of encodings, based on the operational semantics, and then discusses an alternative approach based on measure theory. Section 6 demonstrates the use of the Monte Carlo method in $\lambda_\bigcirc$. Section 7 describes the translation of $\lambda_\bigcirc$ into Objective CAML. Section 8 presents three applications of $\lambda_\bigcirc$ in robotics and discusses the benefit of implementing probabilistic computations in $\lambda_\bigcirc$. Section 9 discusses related work and Section 10 concludes.

[1]`http://caml.inria.fr`

2. A MOTIVATING EXAMPLE

A *Bayes filter* [Jazwinski 1970] is a popular solution to a wide range of state estimation problems. It estimates states of a dynamic system from a sequence of sensor readings called *actions* and *measurements*, where an action a induces a change to the current state and a measurement m gives information on the current state. At its core, a Bayes filter computes a probability distribution Bel of the current state according to the following update equations:

$$Bel(s) \leftarrow \int \mathcal{A}(s|a,s')Bel(s')ds' \quad (1)$$

$$Bel(s) \leftarrow \eta\mathcal{P}(m|s)Bel(s) \quad (2)$$

$\mathcal{A}(s|a,s')$ is the probability that the system transitions to state s after taking action a in another state s', $\mathcal{P}(m|s)$ the probability of measurement m in state s, and η a normalizing constant ensuring $\int Bel(s)ds = 1.0$. As $\leftarrow$ denotes assignment rather than equality, Bel can be thought of as a mutable variable governed by the two update equations, rather than a fixed solution to a pair of recursive equations. The order of applying the update equations is determined by the incoming sequence of sensor readings: each action triggers the update equation (1), and each measurement the update equation (2). In this way, the Bays filter maintains Bel to estimate the current state of the system.

The update equations (1) and (2) are formulated at the level of probability distributions in the sense that they do not assume a particular representation scheme. Unfortunately they are difficult to implement if Bel is allowed to be an arbitrary probability distribution. When it comes to implementation, therefore, we usually simplify the update equations by making additional assumptions on the system or choosing a specific representation scheme. For example, with the assumption that Bel is a Gaussian distribution, we obtain a variant of the Bayes filter called a *Kalman filter* [Welch and Bishop 1995]. If Bel is approximated with a set of samples, we obtain another variant called a *particle filter* [Doucet et al. 2001].

Even these variants of the Bayes filter are, however, not trivial to implement in conventional languages. For example, a Kalman filter requires various matrix operations including matrix inversion. A particle filter manipulates weights associated with individual samples, which often results in complicated code. Since conventional languages do not treat probability distributions as primitive datatypes, it is also difficult to figure out the intended meaning of the code, namely the update equations for the Bayes filter.

An alternative approach is to use an existing probabilistic language after discretizing all probability distributions. This idea is appealing in theory, but impractical for two reasons. First, given a probability distribution, it may not be easy to choose an appropriate subset of its support upon which discretization is performed. For example, in order to discretize a Gaussian distribution (whose support is $(-\infty, \infty)$), we need to choose a threshold for probabilities so that discretization is confined to an interval of finite length; for an arbitrary probability distribution, such a threshold can be computed only by examining its entire probability domain. Even when the subset of its support is fixed in advance, the process of discretization may incur a considerable amount of programming. For example, Fox *et al.* [Fox et al. 1999] develop two non-trivial techniques (specific to their applications) for

type	A, B	$::=$	$A \rightarrow A \mid A \times A \mid \bigcirc A \mid \mathsf{real}$
term	M, N	$::=$	$x \mid \lambda x\!:\!A.\, M \mid M\ M \mid (M, M) \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid$ $\mathsf{fix}\ x\!:\!A.\, M \mid \mathsf{prob}\ E \mid r$
expression	E, F	$::=$	$M \mid \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E \mid \mathcal{S}$
value/sample	V	$::=$	$\lambda x\!:\!A.\, M \mid (V, V) \mid \mathsf{prob}\ E \mid r$
real number	r		
sampling sequence	ω	$::=$	$r_1 r_2 \cdots r_i \cdots \quad \textit{where } r_i \in (0.0, 1.0]$
typing context	Γ	$::=$	$\cdot \mid \Gamma, x : A$

Fig. 1. Abstract syntax for $\lambda_{\bigcirc}$.

the sole purpose of efficiently manipulating discretized probability distributions. Second some probability distributions cannot be discretized in any meaningful way. An example is probability distributions over probability distributions or functions, which do occur in real applications (Section 8 presents such an example).

If we had a probabilistic language that supports all kinds of probability distributions without drawing a syntactic or semantic distinction, we could implement the update equations with much less effort. $\lambda_{\bigcirc}$ is a probabilistic language designed with these goals in mind.

3. PROBABILISTIC LANGUAGE $\lambda_{\bigcirc}$

In this section, we develop our probabilistic language $\lambda_{\bigcirc}$. We exploit the fact that generalized sampling functions, mapping $(0.0, 1.0]^\infty$ to $\mathcal{D} \times (0.0, 1.0]^\infty$ for a probability domain $\mathcal{D}$, form a state monad [Moggi 1989; 1991] whose set of states is $(0.0, 1.0]^\infty$ [Ramsey and Pfeffer 2002], and use a monadic syntax for probabilistic computations in $\lambda_{\bigcirc}$. The end result of using a monadic syntax for probabilistic computations is that it is straightforward to interpret probabilistic computations in terms of sampling functions.

3.1 Syntax and type system

As the linguistic framework of $\lambda_{\bigcirc}$, we use the monadic metalanguage of Pfenning and Davies [Pfenning and Davies 2001]. It is a reformulation of Moggi's monadic metalanguage λ_{ml} [Moggi 1991], following Martin-Löf's methodology of distinguishing judgments from propositions [Martin-Löf 1996]. It augments the lambda calculus, consisting of *terms*, with a separate syntactic category, consisting of *expressions* in a monadic syntax. In the case of $\lambda_{\bigcirc}$, terms denote regular values and expressions denote probabilistic computations in the sense that under its operational semantics, a term reduces to a unique regular value and an expression reduces to a probabilistically chosen sample. We say that a term *evaluates* to a value and an expression *computes* to a sample.

Figure 1 shows the abstract syntax for $\lambda_{\bigcirc}$. We use x for variables. $\lambda x\!:\!A.\, M$ is a lambda abstraction, and $M\ N$ is an application term. (M, N) is a product term, and $\mathsf{fst}\ M$ and $\mathsf{snd}\ M$ are projection terms; we include these terms to support joint distributions. $\mathsf{fix}\ x\!:\!A.\, M$ is a fixed point construct for recursive evaluations. A *probability term* $\mathsf{prob}\ E$ encapsulates expression E; it is a first-class value denoting a probability distribution. r is a real number.

There are three kinds of expressions: term M, *bind expression* $\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E$,

$$\frac{}{\Gamma, x : A \vdash x : A}\ \mathsf{Hyp} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x{:}A.\, M : A \to B}\ \mathsf{Lam} \qquad \frac{\Gamma \vdash M_1 : A \to B \quad \Gamma \vdash M_2 : A}{\Gamma \vdash M_1\ M_2 : B}\ \mathsf{App}$$

$$\frac{\Gamma \vdash M_1 : A_1 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash (M_1, M_2) : A_1 \times A_2}\ \mathsf{Prod} \qquad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \mathsf{fst}\ M : A_1}\ \mathsf{Fst} \qquad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \mathsf{snd}\ M : A_2}\ \mathsf{Snd}$$

$$\frac{\Gamma, x : A \vdash M : A}{\Gamma \vdash \mathsf{fix}\ x{:}A.\, M : A}\ \mathsf{Fix} \qquad \frac{\Gamma \vdash E \div A}{\Gamma \vdash \mathsf{prob}\ E : \bigcirc A}\ \mathsf{Prob} \qquad \frac{}{\Gamma \vdash r : \mathsf{real}}\ \mathsf{Real}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M \div A}\ \mathsf{Term} \qquad \frac{\Gamma \vdash M : \bigcirc A \quad \Gamma, x : A \vdash E \div B}{\Gamma \vdash \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E \div B}\ \mathsf{Bind} \qquad \frac{}{\Gamma \vdash \mathcal{S} \div \mathsf{real}}\ \mathsf{Sampling}$$

Fig. 2. Typing rules of $\lambda_{\bigcirc}$.

and *sampling expression* $\mathcal{S}$. As an expression, M denotes a (degenerate) probabilistic computation that returns the result of evaluating M. sample x from M in E sequences two probabilistic computations (if M evaluates to a probability term). $\mathcal{S}$ consumes a random number in a *sampling sequence*, an infinite sequence of random numbers drawn independently from $U(0.0, 1.0]$.

The type system employs two kinds of typing judgments:

— term typing judgment $\Gamma \vdash M : A$, meaning that M evaluates to a value of type A under typing context Γ.

— expression typing judgment $\Gamma \vdash E \div A$, meaning that E computes to a sample of type A under typing context Γ.

A typing context Γ is a set of bindings $x : A$. Figure 2 shows the typing rules of $\lambda_{\bigcirc}$. The rule Prob is the introduction rule for the type constructor $\bigcirc$; it means that type $\bigcirc A$ denotes probability distributions over type A. The rule Bind is the elimination rule for the type constructor $\bigcirc$. The rule Term means that every term converts into a probabilistic computation that involves no probabilistic choice. The rule Real shows that real is the type of real numbers. A sampling expression $\mathcal{S}$ has also type real, as shown in the rule Sampling, because it computes to a real number. All the remaining rules are standard.

3.2 Operational semantics

Since $\lambda_{\bigcirc}$ draws a syntactic distinction between regular values and probabilistic computations, its operational semantics needs two kinds of judgments:

— term evaluation judgment $M \rightharpoonup V$, meaning that term M evaluates to value V.

— expression computation judgment $E\ @\ \omega \rightharpoondown V\ @\ \omega'$, meaning that expression E with sampling sequence ω computes to sample V with remaining sampling sequence ω'. Conceptually $E\ @\ \omega \rightharpoondown V\ @\ \omega'$ consumes random numbers in $\omega - \omega'$.

For term evaluations, we introduce a term reduction $M \mapsto_{\mathsf{t}} N$ in a call-by-value discipline. We could have equally chosen call-by-name or call-by-need, but $\lambda_{\bigcirc}$ is intended to be embedded in Objective CAML and hence we choose call-by-value for pragmatic reasons. We identify $M \mapsto_{\mathsf{t}}^{*} V$ with $M \rightharpoonup V$, where $\mapsto_{\mathsf{t}}^{*}$ is the reflexive and transitive closure of $\mapsto_{\mathsf{t}}$. For expression computations, we introduce an expression reduction $E\ @\ \omega \mapsto_{\mathsf{e}} F\ @\ \omega'$ such that $E\ @\ \omega \mapsto_{\mathsf{e}}^{*} V\ @\ \omega'$ is identified

$$\frac{M \mapsto_{\mathsf{t}} M'}{M\ N \mapsto_{\mathsf{t}} M'\ N}\ T_{\beta_L} \qquad \frac{N \mapsto_{\mathsf{t}} N'}{(\lambda x\!:\!A.\,M)\ N \mapsto_{\mathsf{t}} (\lambda x\!:\!A.\,M)\ N'}\ T_{\beta_R} \qquad \frac{}{(\lambda x\!:\!A.\,M)\ V \mapsto_{\mathsf{t}} [V/x]M}\ T_{\beta_V}$$

$$\frac{M \mapsto_{\mathsf{t}} M'}{(M, N) \mapsto_{\mathsf{t}} (M', N)}\ T_{P_L} \qquad \frac{N \mapsto_{\mathsf{t}} N'}{(V, N) \mapsto_{\mathsf{t}} (V, N')}\ T_{P_R} \qquad \frac{M \mapsto_{\mathsf{t}} N}{\mathsf{fst}\ M \mapsto_{\mathsf{t}} \mathsf{fst}\ N}\ T_{Fst} \qquad \frac{}{\mathsf{fst}\ (V, V') \mapsto_{\mathsf{t}} V}\ T_{Fst'}$$

$$\frac{M \mapsto_{\mathsf{t}} N}{\mathsf{snd}\ M \mapsto_{\mathsf{t}} \mathsf{snd}\ N}\ T_{Snd} \qquad \frac{}{\mathsf{snd}\ (V, V') \mapsto_{\mathsf{t}} V'}\ T_{Snd'} \qquad \frac{}{\mathsf{fix}\ x\!:\!A.\,M \mapsto_{\mathsf{t}} [\mathsf{fix}\ x\!:\!A.\,M/x]M}\ T_{Fix}$$

$$\frac{M \mapsto_{\mathsf{t}} N}{M \ @\ \omega \mapsto_{\mathsf{e}} N \ @\ \omega}\ E_{Term} \qquad \frac{M \mapsto_{\mathsf{t}} N}{\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ F \ @\ \omega \mapsto_{\mathsf{e}} \mathsf{sample}\ x\ \mathsf{from}\ N\ \mathsf{in}\ F \ @\ \omega}\ E_{Bind}$$

$$\frac{E \ @\ \omega \mapsto_{\mathsf{e}} E' \ @\ \omega'}{\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E\ \mathsf{in}\ F \ @\ \omega \mapsto_{\mathsf{e}} \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E'\ \mathsf{in}\ F \ @\ \omega'}\ E_{BindR}$$

$$\frac{}{\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ V\ \mathsf{in}\ F \ @\ \omega \mapsto_{\mathsf{e}} [V/x]F \ @\ \omega}\ E_{BindV} \qquad \frac{}{\mathcal{S} \ @\ r\omega \mapsto_{\mathsf{e}} r \ @\ \omega}\ \mathit{Sampling}$$

Fig. 3. Operational semantics of $\lambda_{\bigcirc}$.

with $E \ @\ \omega \rightharpoondown V \ @\ \omega'$, where $\mapsto_{\mathsf{e}}^{*}$ is the reflexive and transitive closure of $\mapsto_{\mathsf{e}}$. Both reductions use capture-avoiding term substitutions $[M/x]N$ and $[M/x]E$ defined in a standard way.

Figure 3 shows the reduction rules in the operational semantics of $\lambda_{\bigcirc}$. Expression reductions may invoke term reductions (*e.g.*, to reduce M in $\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E$). The rules E_{BindR} and E_{BindV} mean that given a bind expression $\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E\ \mathsf{in}\ F$, we finish computing E before substituting a value for x in F. Note that like a term evaluation, an expression computation itself is deterministic; it is only when we vary sampling sequences that an expression exhibits probabilistic behavior.

An expression computation $E \ @\ \omega \mapsto_{\mathsf{e}}^{*} V \ @\ \omega'$ means that E takes a sampling sequence ω, consumes a finite prefix of ω in order, and returns a sample V with the remaining sampling sequence ω':

PROPOSITION 3.1. *If $E \ @\ \omega \mapsto_{\mathsf{e}}^{*} V \ @\ \omega'$, then $\omega = r_1 r_2 \cdots r_n \omega'$ $(n \geq 0)$ where*

$$E \ @\ \omega \mapsto_{\mathsf{e}}^{*} \cdots \mapsto_{\mathsf{e}}^{*} E_i \ @\ r_{i+1} \cdots r_n \omega' \mapsto_{\mathsf{e}}^{*} \cdots \mapsto_{\mathsf{e}}^{*} E_n \ @\ \omega' \mapsto_{\mathsf{e}}^{*} V \ @\ \omega'$$

for a sequence of expressions $E_1, \cdots, E_n$.

Thus an expression computation coincides with the operational description of a sampling function when applied to a sampling sequence, which implies that an expression specifies a sampling function. (Here we use a generalized notion of sampling function mapping $(0.0, 1.0]^{\infty}$ to $A \times (0.0, 1.0]^{\infty}$ for a certain type A.)

The type safety of $\lambda_{\bigcirc}$ consists of two properties: type preservation and progress. The proof of type preservation requires a substitution lemma, and the proof of progress requires a canonical forms lemma.

LEMMA 3.2 SUBSTITUTION.
If $\Gamma \vdash M : A$ and $\Gamma, x : A \vdash N : B$, then $\Gamma \vdash [M/x]N : B$.
If $\Gamma \vdash M : A$ and $\Gamma, x : A \vdash E \div B$, then $\Gamma \vdash [M/x]E \div B$.

PROOF. By simultaneous induction on the structure of N and E. □

THEOREM 3.3 TYPE PRESERVATION.

If $M \mapsto_{\mathsf{t}} N$ and $\cdot \vdash M : A$, then $\cdot \vdash N : A$.
If $E \ @\ \omega \mapsto_{\mathsf{e}} F \ @\ \omega'$ and $\cdot \vdash E \div A$, then $\cdot \vdash F \div A$.

Proof. By induction on the structure of M and E. □

Lemma 3.4 Canonical forms. *If $\cdot \vdash V : A$, then:*
(1) if $A = A_1 \rightarrow A_2$, then $V = \lambda x\!:\!A_1.\, M$.
(2) if $A = A_1 \times A_2$, then $V = (V_1, V_2)$.
(3) if $A = \bigcirc A'$, then $V = \mathsf{prob}\ E$.
(4) if $A = \mathsf{real}$, then $V = r$.

Proof. By case analysis of A. □

Theorem 3.5 Progress.
*If $\cdot \vdash M : A$, then either M is a value (*i.e.*, $M = V$), or there exists a unique term N such that $M \mapsto_{\mathsf{t}} N$.*
*If $\cdot \vdash E \div A$, then either E is a sample (*i.e.*, $E = V$), or for any sampling sequence ω, there exist a unique expression F and a unique sampling sequence ω' such that $E \ @\ \omega \mapsto_{\mathsf{e}} F \ @\ \omega'$.*

Proof. For the first clause, we show by induction on the structure of M. For the second clause, we show by induction on the structure of E; we use the result from the first clause. □

Since terms are special cases of expressions, an expression computation may involve smaller term evaluations by the rule E_{Term}. The converse is not the case, however: there is no way for a term evaluation to initiate an expression computation or for an expression computation to return its result back to a term evaluation. As a result, we can write terms of type $\bigcirc A$ denoting probability distributions, but never "observe" probabilistic computations (*e.g.*, generating samples, calculating means, or calculating the probability of an event) during term evaluations. Section 6 develops two additional constructs, expectation (for the expectation query) and bayes (for the Bayes operation), which enable us to "observe" probabilistic computations during term evaluations, similarly to unsafePerformIO [Peyton Jones and Wadler 1993] and runST [Launchbury and Peyton Jones 1995] of Haskell.

3.3 Fixed point construct for expressions

In $\lambda_{\bigcirc}$, expressions specify non-recursive sampling functions. In order to be able to directly specify recursive sampling functions as well (which are useful in creating such probability distributions as geometric distributions), we introduce an *expression variable* $\mathbf{x}$ and an *expression fixed point construct* $\mathsf{efix}\ \mathbf{x}\!\div\! A.\, E$; a new form of binding $\mathbf{x} \div A$ for expression variables is used in typing contexts:

$$\begin{array}{llcl} \text{expression} & E & ::= & \cdots \mid \mathbf{x} \mid \mathsf{efix}\ \mathbf{x}\!\div\! A.\, E \\ \text{typing context} & \Gamma & ::= & \cdots \mid \Gamma, \mathbf{x} \div A \end{array}$$

New typing rules and reduction rule are as follows:

$$\frac{}{\Gamma, \mathbf{x} \div A \vdash \mathbf{x} \div A}\ \mathsf{Evar} \qquad \frac{\Gamma, \mathbf{x} \div A \vdash E \div A}{\Gamma \vdash \mathsf{efix}\ \mathbf{x}\!\div\! A.\, E \div A}\ \mathsf{Efix}$$

$$\frac{}{\mathsf{efix}\ \mathbf{x}\!\div\! A.\, E \ @\ \omega \mapsto_{\mathsf{e}} [\mathsf{efix}\ \mathbf{x}\!\div\! A.\, E/\mathbf{x}]E \ @\ \omega}\ \mathit{Efix}$$

In the rule *Efix*, $[\mathsf{efix}\ \mathbf{x} \div A.\, E/\mathbf{x}]E$ denotes a capture-avoiding substitution of $\mathsf{efix}\ \mathbf{x} \div A.\, E$ for expression variable $\mathbf{x}$. Thus $\mathsf{efix}\ \mathbf{x} \div A.\, E$ behaves like term fixed point constructs except that it unrolls itself by substituting an expression for an expression variable, instead of a term for an ordinary variable.

Expression fixed point constructs are syntactic sugar as they can be simulated with fixed point constructs for terms. The intuition is that we first build a term fixed point construct M of type $\bigcirc A$ and then convert it into an expression $\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ x$, which denotes a recursive computation. To simulate expression fixed point constructs, we define a function $(\cdot)^\star$ which translates $(\mathsf{efix}\ \mathbf{x} \div A.\, E)^\star$ into:

$$\mathsf{sample}\ y_r\ \mathsf{from}\ \mathsf{fix}\ x_p\!:\!\bigcirc A.\, \mathsf{prob}\ [\mathsf{sample}\ y_v\ \mathsf{from}\ x_p\ \mathsf{in}\ y_v/\mathbf{x}]E^\star\ \mathsf{in}\ y_r$$

That is, we introduce a variable x_p to encapsulate $\mathsf{efix}\ \mathbf{x} \div A.\, E$ and expand $\mathbf{x}$ to a bind expression $\mathsf{sample}\ y_v\ \mathsf{from}\ x_p\ \mathsf{in}\ y_v$. Using the syntactic sugar $\mathsf{unprob}\ M$ defined as $\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ x$ (to be introduced in Section 4), we can rewrite $(\mathsf{efix}\ \mathbf{x} \div A.\, E)^\star$ as follows:

$$\mathsf{unprob}\ \mathsf{fix}\ x_p\!:\!\bigcirc A.\, \mathsf{prob}\ [\mathsf{unprob}\ x_p/\mathbf{x}]E^\star$$

The translation of other terms and expressions is structural:

$$\begin{array}{rcl}
x^\star & = & x \\
(\lambda x\!:\!A.\, M)^\star & = & \lambda x\!:\!A.\, M^\star \\
(M_1\ M_2)^\star & = & {M_1}^\star\ {M_2}^\star \\
(\mathsf{prob}\ E)^\star & = & \mathsf{prob}\ E^\star \\
(\mathsf{fix}\ x\!:\!A.\, M)^\star & = & \mathsf{fix}\ x\!:\!A.\, M^\star \\
(\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E)^\star & = & \mathsf{sample}\ x\ \mathsf{from}\ M^\star\ \mathsf{in}\ E^\star \\
\mathbf{x}^\star & = & \mathbf{x}
\end{array}$$

Proposition 3.6 shows that when translated via the function $(\cdot)^\star$, the typing rules Evar and Efix are sound with respect to the original type system (without the rules Evar and Efix). See Appendix A.1 for a proof.

Proposition 3.6.
If $\Gamma \vdash M : A$, *then* $\Gamma \vdash M^\star : A$.
If $\Gamma \vdash E \div A$, *then* $\Gamma \vdash E^\star \div A$.

Since $M^\star$ and $E^\star$ do not contain expression fixed point constructs, the rule Efix is not used in $\Gamma \vdash M^\star : A$ and $\Gamma \vdash E^\star \div A$. Neither is the rule Evar used unless M or E contains free expression variables. Therefore, given a term or expression with no free expression variable, the function $(\cdot)^\star$ returns another term or expression of the same type which does not need the rules Evar and Efix.

Propositions 3.7 and 3.8 show that the reduction rule *Efix* is sound and complete with respect to the operational semantics of $\lambda_\bigcirc$. We use the fact that the computation of $E^\star$ does not require the rule *Efix*. See Appendix A.2 for proofs and the definition of an equivalence relation $\equiv_\mathsf{e}$.

Proposition 3.7.
If $E\ @\ \omega \mapsto_\mathsf{e} F\ @\ \omega'$ *with the rule Efix, then* $E^\star\ @\ \omega \mapsto_\mathsf{e} F'\ @\ \omega'$ *and* $F' \equiv_\mathsf{e} F^\star$.

Proposition 3.8.
If $E^\star\ @\ \omega \mapsto_\mathsf{e} F'\ @\ \omega'$, *then there exists* F *such that* $F' \equiv_\mathsf{e} F^\star$ *and* $E\ @\ \omega \mapsto_\mathsf{e} F\ @\ \omega'$.

3.4 Distinguishing terms and expressions

The decision to distinguish terms and expressions in $\lambda_{\bigcirc}$ is a consequence of using a monadic syntax for probabilistic computations, or equivalently, designing $\lambda_{\bigcirc}$ as a monadic language that regards probabilistic computations as a particular case of computational effects. A typical monadic language, while disguised as a language with a single syntactic category, actually has two sublanguages: a functional sublanguage and a monadic sublanguage. For example, Peyton Jones [Peyton Jones 2001] clarifies the distinction between the two with a semantics for Haskell which is stratified into two levels: an *inner* (denotational) semantics for the functional sublanguage and an *outer* (transition) semantics for the monadic sublanguage. The linguistic framework of $\lambda_{\bigcirc}$ (namely the monadic metalanguage of Pfenning and Davies [Pfenning and Davies 2001]) reformulates Moggi's monadic metalanguage λ_{ml} [Moggi 1991] by making the distinction syntactically explicit.

The *syntactic* distinction between terms and expressions in $\lambda_{\bigcirc}$ is optional in that the grammar does not need to distinguish terms as a separate non-terminal; see [Park 2003] for a probabilistic language in which sampling functions serve as the mathematical basis, but no syntactic distinction is drawn between terms and expressions, *i.e.*, everything is an expression. On the other hand, the *semantic* distinction in $\lambda_{\bigcirc}$, both statically (in the form of term and expression typing judgments) and dynamically (in the form of evaluation and computation judgments), appears to be desirable because of the dominant role of regular values in probabilistic computations. (For example, we do not wish to use a point-mass probability distribution for an integer index in a loop.) [Park 2003] uses a type system with subtypes and intersection types to distinguish expressions denoting regular values from expressions denoting probabilistic computations.

$\lambda_{\bigcirc}$ is a conservative extension of a conventional language because terms constitute a conventional language of their own. By Theorem 3.5, term evaluations are always deterministic and we need only terms when writing deterministic programs. As a separate syntactic category, expressions provide a framework for probabilistic computations that abstracts from the definition of terms. In this regard, expressions constitute a *probabilistic* language parameterized by a *base* language consisting of terms, along the line of Leroy [Leroy 2000] who presents an implementation of a module system parameterized by a base language and its typechecker. Thus adding new term constructs does not change the definition of expressions; changing the operational semantics for terms (*e.g.*, from call-by-value to call-by-name) does not affect type safety of $\lambda_{\bigcirc}$ as long as type safety for terms is not violated.

When programming in $\lambda_{\bigcirc}$, the syntactic distinction between terms and expressions aids us in deciding which of deterministic evaluations and probabilistic computations we should focus on. In the next section, we show how to encode various probability distributions and further investigate properties of $\lambda_{\bigcirc}$.

4. EXAMPLES

When encoding a probability distribution in $\lambda_{\bigcirc}$, we naturally concentrate on a method of generating samples, rather than a method of calculating the probability assigned to each event. If a process for generating samples is known, we simply translate it in $\lambda_{\bigcirc}$. If, however, the probability distribution is defined in terms of a probability measure or an equivalent, we may not always derive a sampling function in a mechanical manner. Instead we have to exploit its unique properties to devise a sampling function.

Below we show examples of encoding various probability distributions in $\lambda_{\bigcirc}$. These examples demonstrate three properties of $\lambda_{\bigcirc}$: a unified representation scheme for probability distributions, rich expressiveness, and high versatility in encoding probability distributions. The sampling methods used in the examples are all found in simulation theory. Common sampling methods developed in simulation theory are easy to translate in $\lambda_{\bigcirc}$, which thus serves as a programming language allowing sampling methods developed in simulation theory to be formally expressed in a concise and readable fashion. We refer the reader to the literature on simulation theory for sampling methods (*e.g.*, see Chapter 5 of [Bratley et al. 1996] for a quick introduction to sampling methods and [Devroye 1986] for a comprehensive introduction with detailed proofs); Section 5 shows how to transcribe correctness proofs of sampling methods in simulation theory into correctness proofs of encodings in $\lambda_{\bigcirc}$.

We assume primitive types int and bool (with boolean values True and False), arithmetic and comparison operators, and a conditional term construct if M then N_1 else N_2. We also assume standard let-binding, recursive let rec-binding, and pattern matching when it is convenient for the examples.[2] We use the following syntactic sugar for expressions:

$$\begin{array}{rcl} \mathsf{unprob}\ M & \equiv & \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ x \\ \mathsf{eif}\ M\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 & \equiv & \mathsf{unprob}\ (\mathsf{if}\ M\ \mathsf{then}\ \mathsf{prob}\ E_1\ \mathsf{else}\ \mathsf{prob}\ E_2) \end{array}$$

unprob M chooses a sample from the probability distribution denoted by M (we choose the keyword unprob to suggest that it does the opposite of what prob does.) eif M then E_1 else E_2 branches to either E_1 or E_2 depending on the result of evaluating M.

Unified representation scheme

$\lambda_{\bigcirc}$ provides a unified representation scheme for probability distributions. While its type system distinguishes between different probability domains, its operational semantics does not distinguish between different kinds of probability distributions, such as discrete, continuous, or neither. We show examples for these three cases.

We encode a point-mass distribution centered on x (of type real) as follows:

$$\mathsf{let}\ \mathit{point\_mass} = \lambda x{:}\mathsf{real}.\ \mathsf{prob}\ x$$

We encode a Bernoulli distribution over type bool with parameter p as follows:

$$\begin{array}{l} \mathsf{let}\ \mathit{bernoulli} = \lambda p{:}\mathsf{real}.\ \mathsf{prob}\ \ \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ \mathcal{S}\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\qquad\qquad x \le p \end{array}$$

[2]If type inference and polymorphism are ignored, let-binding and recursive let rec-binding may be interpreted as follows, where _ is a wildcard pattern for types:

$$\begin{array}{rcl} \mathsf{let}\ x = M\ \mathsf{in}\ N & \equiv & (\lambda x{:}\_.\ N)\ M \\ \mathsf{let\ rec}\ x = M\ \mathsf{in}\ N & \equiv & \mathsf{let}\ x = \mathsf{fix}\ x{:}\_.\ M\ \mathsf{in}\ N \end{array}$$

bernoulli can be thought of as a binary choice construct. It is expressive enough to specify any discrete distribution with finite support. In fact, *bernoulli* 0.5 suffices to specify all such probability distributions, since it is capable of simulating a binary choice construct [Gill 1977] (if the probability assigned to each element in the domain is computable).

As an example of continuous distribution, we encode a uniform distribution over a real interval $(a, b]$ by exploiting the definition of the sampling expression:

```
let $uniform = \lambda a$:real. $\lambda b$:real. prob  sample $x$ from prob $\mathcal{S}$ in
                                           $a + x * (b - a)$
```

We also encode a combination of a point-mass distribution and a uniform distribution over the same domain, which is neither a discrete distribution nor a continuous distribution:

```
let $choice = \lambda p$:real. $\lambda x$:$\bigcirc$real. $\lambda y$:$\bigcirc$real. prob  sample $x$ from prob $\mathcal{S}$ in
                                                    eif $x \leq p$ then unprob $x$ else unprob $y$
let $point\_uniform = choice$ 0.5 ($point\_mass$ 0.0) ($uniform$ 0.5 1.0)
```

Here *choice* is a combinator generalizing a binary choice construct: $choice\ p\ P_1\ P_2$ generates a sample from P_1 with probability p and from P_2 with probability $1.0 - p$.

Rich expressiveness

We have seen above that the expressive power of $\lambda_\bigcirc$ covers discrete distributions with finite support. It turns out that $\lambda_\bigcirc$ allows us to encode discrete distributions with countable support as well, which follows from the fact that $\lambda_\bigcirc$ can express the sum of a countable set of weighted probability distributions. Consider such a countable set $\{(w_i, \mathsf{prob}\ E_i) | 1 \leq i\}$ with $\sum_{i=1}^{\infty} w_i = 1.0$, which assigns a weight (or a probability) w_i to the probability distribution denoted by prob E_i. We represent it as a function f of type int$\rightarrow$(real $\times \bigcirc A$) such that $f\ i \rightharpoonup (w_i, \mathsf{prob}\ E_i)$, and encode its sum as follows:

```
let $countable\_sum = \lambda f$:int$\rightarrow$(real $\times \bigcirc A$).
   let rec $scan = \lambda n$:int. $\lambda r$:real.
      if $r -$ fst $(f\ n) \leq 0.0$ then snd $(f\ n)$ else $scan\ (n+1)\ (r -$ fst $(f\ n))$
      in
   prob  sample $x$ from prob $\mathcal{S}$ in
         sample $y$ from $scan$ 1 $x$ in
         $y$
```

That is, we evaluate $scan\ 1\ x$ to locate a probability distribution corresponding to a random number x, and then generate a sample y from it. In the special case where each prob E_i denotes a point-mass distribution, we obtain a discrete distribution with countable support.

The expressive power of $\lambda_\bigcirc$ is further increased by various sampling methods borrowed from simulation theory. All sampling methods developed in simulation theory hinge on the ability to generate samples from uniform distributions over real intervals, which is trivial to achieve in $\lambda_\bigcirc$. Here are a couple of general sampling methods applicable to a wide range of probability distributions:

—The *inverse transform method* uses the inverse of a cumulative distribution function as a sampling function. It is extensible to n-dimensional real space and requires only as many random numbers as the dimension of the probability domain.

—The *rejection method* requires knowledge of a probability density function, and generates a sample from a (continuous) probability distribution by repeatedly generating samples from other probability distributions until they satisfy a certain condition. It is particularly useful when no cumulative distribution function is known in closed form, but a probability density function is known (*e.g.*, Gaussian distributions).

We now demonstrate the expressive power of $\lambda_{\bigcirc}$ with a number of examples.

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

```
let $binomial = \lambda p{:}\mathsf{real}.\, \lambda n_0{:}\mathsf{int}.$
  let $bernoulli_p = bernoulli\ p$ in
  let rec $binomial_p = \lambda n{:}\mathsf{int}.$
    if $n = 0$ then prob $0$
    else prob  sample $x$ from $binomial_p\ (n-1)$ in
               sample $b$ from $bernoulli_p$ in
               if $b$ then $1 + x$ else $x$
    in
  $binomial_p\ n_0$
```

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

A recursive process for generating samples can be translated into a recursive term in $\lambda_{\bigcirc}$. For example, we encode a geometric distribution with parameter p, which is a discrete distribution with infinite support, as follows:

```
let $geometric\_rec = \lambda p{:}\mathsf{real}.$
  let $bernoulli_p = bernoulli\ p$ in
  let rec $geometric =$ prob  sample $b$ from $bernoulli_p$ in
                              eif $b$ then $0$
                              else  sample $x$ from $geometric$ in
                                    $1 + x$
    in
  $geometric$
```

Here we use a recursive term *geometric* of type $\bigcirc$int. Equivalently we can use an expression fixed point construct:

```
let $geometric\_efix = \lambda p{:}\mathsf{real}.$ let $bernoulli_p = bernoulli\ p$ in
                                   prob  efix $\mathbf{geometric} \div \mathsf{int}.$
                                           sample $b$ from $bernoulli_p$ in
                                           eif $b$ then $0$
                                           else  sample $x$ from prob $\mathbf{geometric}$ in
                                                 $1 + x$
```

We encode an exponential distribution by the inverse transform method:

```
let $exponential_{1.0}$ = prob  sample $x$ from prob $\mathcal{S}$ in
                                $-$log $x$
```

The rejection method can be implemented with a recursive term. For example, we encode a Gaussian distribution with mean m and variance σ^2 by the rejection method with respect to exponential distributions:

```
let $bernoulli_{0.5}$ = $bernoulli$ 0.5
let $gaussian\_rejection$ = $\lambda m$ : real. $\lambda \sigma$ : real.
   let rec $gaussian$ = prob  sample $y_1$ from $exponential_{1.0}$ in
                              sample $y_2$ from $exponential_{1.0}$ in
                              eif $y_2 \geq (y_1 - 1.0)^2/2.0$ then
                                 sample $b$ from $bernoulli_{0.5}$ in
                                 if $b$ then $m + \sigma * y_1$ else $m - \sigma * y_1$
                              else unprob $gaussian$
   in
   $gaussian$
```

We encode the joint distribution between two independent probability distributions using a product term. If M_P denotes $P(x)$ and M_Q denotes $Q(y)$, the following term denotes the joint distribution $Prob(x, y) \propto P(x)Q(y)$:

```
prob  sample $x$ from $M_P$ in
      sample $y$ from $M_Q$ in
      $(x, y)$
```

For the joint distribution between two interdependent probability distributions, we use a conditional probability, which we represent as a lambda abstraction taking a regular value and returning a probability distribution. If M_P denotes $P(x)$ and M_Q denotes a conditional probability $Q(y|x)$, the following term denotes the joint distribution $Prob(x, y) \propto P(x)Q(y|x)$:

```
prob  sample $x$ from $M_P$ in
      sample $y$ from $M_Q$ $x$ in
      $(x, y)$
```

By returning y instead of (x, y), we compute the integration $Prob(y) = \int P(x)Q(y|x)dx$:

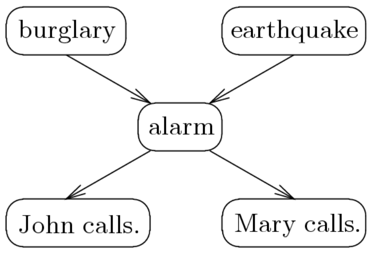
```
prob  sample $x$ from $M_P$ in
      sample $y$ from $M_Q$ $x$ in
      $y$
```

Due to lack of semantic constraints on sampling functions, we can specify probability distributions over unusual domains such as infinite data structures (*e.g.*, trees), function spaces, cyclic spaces (*e.g.*, angular values), and even probability distributions themselves. For example, it is straightforward to combine two probability distributions over angular values to compute the probability distribution over sums of their respective angular values:

```
let $add\_angle = \lambda a_1 : \bigcirc$real. $\lambda a_2 : \bigcirc$real. prob  sample $s_1$ from $a_1$ in
                                                  sample $s_2$ from $a_2$ in
                                                  $(s_1 + s_2)$ mod $(2.0 * \pi)$
```

With the modulo operation mod, we take into account the fact that an angle θ is identified with $\theta + 2\pi$.

As a simple application, we implement a belief network [Russell and Norvig 1995]:



We assume that $P_{alarm|burglary}$ denotes the probability distribution that the alarm goes off when a burglary happens; other variables of the form $P_{\cdot|\cdot}$ are interpreted in a similar way.

```
let $alarm = \lambda(burglary, earthquake)$ : bool × bool.
    if $burglary$ then $P_{alarm|burglary}$
    else if $earthquake$ then $P_{alarm|\neg burglary \wedge earthquake}$
    else $P_{alarm|\neg burglary \wedge \neg earthquake}$
let $john\_calls = \lambda alarm$ : bool.
    if $alarm$ then $P_{John\_calls|alarm}$
    else $P_{John\_calls|\neg alarm}$
let $mary\_calls = \lambda alarm$ : bool.
    if $alarm$ then $P_{Mary\_calls|alarm}$
    else $P_{Mary\_calls|\neg alarm}$
```

The conditional probabilities *alarm*, *john_calls*, and *mary_calls* do not answer any query on the belief network and only describe its structure. In order to answer a specific query, we have to implement a corresponding probability distribution. For example, in order to answer "What is the probability $p_{Mary\_calls|John\_calls}$ that Mary calls when John calls?", we use $Q_{Mary\_calls|John\_calls}$ below, which essentially implements logic sampling [Henrion 1988]:

```
let rec $Q_{Mary\_calls|John\_calls}$ = prob  sample $b$ from $P_{burglary}$ in
                                     sample $e$ from $P_{earthquake}$ in
                                     sample $a$ from $alarm$ $(b, e)$ in
                                     sample $j$ from $john\_calls$ $a$ in
                                     sample $m$ from $mary\_calls$ $a$ in
                                     eif $j$ then $m$ else unprob $Q_{Mary\_calls|John\_calls}$
  in
$Q_{Mary\_calls|John\_calls}$
```

$P_{burglary}$ denotes the probability distribution that a burglary happens, and $P_{earthquake}$ the probability distribution that an earthquake happens. Then the mean of $Q_{Mary\_calls|John\_calls}$ gives $p_{Mary\_calls|John\_calls}$. We will see how to calculate $p_{Mary\_calls|John\_calls}$ in Section 6.

We can also implement most of the common operations on probability distributions. An exception is the Bayes operation $\sharp$ (which is used in the second update equation of the Bayes filter). For two independent probability distributions P and Q, the Bayes operation $P \sharp Q$ results in a probability distribution R such that $R(x) = \eta P(x)Q(x)$ where η is a normalization constant ensuring $\int R(x)dx = 1.0$; if $P(x)Q(x)$ is zero for every x, then $P \sharp Q$ is undefined. Since it is difficult to achieve a general implementation of $P \sharp Q$, we usually make an additional assumption on P or Q to achieve a specialized implementation. For example, assuming 1) a function p and a constant c such that $p(x) = kP(x) \leq c$ for a certain (unknown) constant k and 2) a probability term q denoting Q, we can implement $P \sharp Q$ by the rejection method:

```
let $bayes\_rejection = \lambda p{:}A \rightarrow \mathsf{real}.\ \lambda c{:}\mathsf{real}.\ \lambda q{:}\bigcirc A.$
  let rec $bayes$ = prob  sample $x$ from $q$ in
                          sample $u$ from prob $\mathcal{S}$ in
                          eif $u < (p\ x)/c$ then $x$ else unprob $bayes$
    in
  $bayes$
```

Note that we do not use k which only indicates that $p(x)$ is proportional to $P(x)$ (and is unknown anyway). Thus we can implement $P \sharp Q$ when In Section 6, we will see another implementation that does not need even c.

High versatility

$\lambda_{\bigcirc}$ allows high versatility in encoding probability distributions: given a probability distribution, we can exploit its unique properties and encode it in many different ways. For example, $exponential_{1.0}$ uses a logarithm function to encode an exponential distribution, but there is also an ingenious method (due to von Neumann) that requires only addition and subtraction operations:

```
let $exponential\_von\_Neumann_{1.0}$ =
  let rec $search = \lambda k{:}\mathsf{real}.\ \lambda u{:}\mathsf{real}.\ \lambda u_1{:}\mathsf{real}.$
    prob  sample $u'$ from prob $\mathcal{S}$ in
          eif $u < u'$ then $k + u_1$
          else
            sample $u$ from prob $\mathcal{S}$ in
            eif $u \leq u'$ then unprob $(search\ k\ u\ u_1)$
            else
              sample $u$ from prob $\mathcal{S}$ in
              unprob $(search\ (k + 1.0)\ u\ u)$
    in
  prob  sample $u$ from prob $\mathcal{S}$ in
        unprob $(search\ 0.0\ u\ u)$
```

The recursive term in $gaussian\_rejection$ consumes at least three random numbers. We can encode a Gaussian distribution with only two random numbers:

```
let $gaussian\_Box\_Muller = \lambda m{:}\mathsf{real}.\,\lambda\sigma{:}\mathsf{real}.$
  prob  sample $u$ from prob $\mathcal{S}$ in
        sample $v$ from prob $\mathcal{S}$ in
        $m + \sigma * \sqrt{-2.0 * \log u} * \cos\,(2.0 * \pi * v)$
```

We can also approximate a Gaussian distribution by exploiting the central limit theorem:

```
let $gaussian\_central = \lambda m{:}\mathsf{real}.\,\lambda\sigma{:}\mathsf{real}.$
  prob  sample $x_1$ from prob $\mathcal{S}$ in
        sample $x_2$ from prob $\mathcal{S}$ in
          ...
        sample $x_{12}$ from prob $\mathcal{S}$ in
        $m + \sigma * (x_1 + x_2 + \cdots + x_{12} - 6.0)$
```

The three examples above serve as evidence of high versatility of $\lambda_\bigcirc$: *the more we know about a probability distribution, the better we can encode it.*

All the examples in this section just rely on our intuition on sampling functions and do not actually prove the correctness of encodings. For example, we still do not know if *bernoulli* indeed encodes a Bernoulli distribution, or equivalently, if the expression in it generates True with probability p. In the next section, we investigate how to formally prove the correctness of encodings.

5. PROVING THE CORRECTNESS OF ENCODINGS

When programming in $\lambda_\bigcirc$, we often ask *"What probability distribution characterizes outcomes of computing a given expression?"* The operational semantics of $\lambda_\bigcirc$ does not directly answer this question because an expression computation returns only a single sample from a certain, yet unknown, probability distribution. Therefore we need a different methodology for interpreting expressions in terms of probability distributions.

We take a simple approach that appeals to our intuition on the meaning of expressions. We write $E \sim \mathit{Prob}$ if outcomes of computing E are distributed according to Prob. To determine Prob from E, we supply an infinite sequence of independent *random variables* from $U(0.0, 1.0]$ and analyze the result of computing E in terms of these random variables. If $E \sim \mathit{Prob}$, then E denotes a probabilistic computation for generating samples from Prob and we regard Prob as the denotation of prob E.

We illustrate the above approach with a few examples. In each example, R_i means the i-th random variable and R_i^∞ means the infinite sequence of random variables beginning from R_i (*i.e.*, $R_i R_{i+1} \cdots$). A random variable is regarded as a value because it represents real numbers in $(0.0, 1.0]$.

As a trivial example, consider prob $\mathcal{S}$. The computation of $\mathcal{S}$ proceeds as follows:

$$\mathcal{S} \;@\; R_1^\infty \mapsto_{\mathsf{e}} R_1 \;@\; R_2^\infty$$

Since the outcome is a random variable from $U(0.0, 1.0]$, we have $\mathcal{S} \sim U(0.0, 1.0]$.

As an example of discrete distribution, consider *bernoulli* p. The expression in it computes as follows:

$$\begin{array}{ll} & \mathsf{sample}\ x\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in}\ x \le p \quad @\ R_1^\infty \\ \mapsto_{\mathsf{e}} & \mathsf{sample}\ x\ \mathsf{from\ prob}\ R_1\ \mathsf{in}\ x \le p \quad @\ R_2^\infty \\ \mapsto_{\mathsf{e}} & R_1 \le p \quad @\ R_2^\infty \\ \mapsto_{\mathsf{e}} & \mathsf{True}\ \ @\ R_2^\infty\ \ \textit{if}\ R_1 \le p; \\ & \mathsf{False}\ \ @\ R_2^\infty\ \ \textit{otherwise}. \end{array}$$

Since R_1 is a random variable from $U(0.0, 1.0]$, the probability of $R_1 \le p$ is p. Thus the outcome is True with probability p and False with probability $1.0 - p$, and *bernoulli* p denotes a Bernoulli distribution with parameter p.

As an example of continuous distribution, consider *uniform* a b. The expression in it computes as follows:

$$\begin{array}{ll} & \mathsf{sample}\ x\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in}\ a + x * (b - a) \quad @\ R_1^\infty \\ \mapsto_{\mathsf{e}}^* & a + R_1 * (b - a) \quad @\ R_2^\infty \end{array}$$

Since we have

$$a + R_1 * (b - a) \in (a_0, b_0] \quad \textit{iff} \quad R_1 \in (\frac{a_0 - a}{b - a}, \frac{b_0 - a}{b - a}],$$

the probability that the outcome lies in $(a_0, b_0]$ is

$$\frac{b_0 - a}{b - a} - \frac{a_0 - a}{b - a} = \frac{b_0 - a_0}{b - a} \propto b_0 - a_0$$

where we assume $(a_0, b_0] \subset (a, b]$. Thus *uniform* a b denotes a uniform distribution over $(a, b]$.

The following proposition shows that *binomial* p n denotes a binomial distribution with parameters p and n, which we write as $\mathit{Binomial}_{p,n}$:

PROPOSITION 5.1. *If* $\mathit{binomial}_p\ n \mapsto_{\mathsf{t}}^* \mathsf{prob}\ E_{p,n}$*, then* $E_{p,n} \sim \mathit{Binomial}_{p,n}$.

PROOF. By induction on n.

Base case $n = 0$. We have $E_{p,n} = 0$. Since $\mathit{Binomial}_{p,n}$ is a point-mass distribution centered on 0, we have $E_{p,n} \sim \mathit{Binomial}_{p,n}$.

Inductive case $n > 0$. The computation of $E_{p,n}$ proceeds as follows:

$$\begin{array}{ll} & \mathsf{sample}\ x\ \mathsf{from}\ \mathit{binomial}_p\ (n-1)\ \mathsf{in} \\ & \mathsf{sample}\ b\ \mathsf{from}\ \mathit{bernoulli}_p\ \mathsf{in} \\ & \mathsf{if}\ b\ \mathsf{then}\ 1 + x\ \mathsf{else}\ x \quad @\ R_1^\infty \\ \mapsto_{\mathsf{e}}^* & \mathsf{sample}\ x\ \mathsf{from\ prob}\ x_{p,n-1}\ \mathsf{in} \\ & \mathsf{sample}\ b\ \mathsf{from}\ \mathit{bernoulli}_p\ \mathsf{in} \\ & \mathsf{if}\ b\ \mathsf{then}\ 1 + x\ \mathsf{else}\ x \quad @\ R_i^\infty \\ \mapsto_{\mathsf{e}}^* & \mathsf{sample}\ b\ \mathsf{from\ prob}\ b_p\ \mathsf{in} \\ & \mathsf{if}\ b\ \mathsf{then}\ 1 + x_{p,n-1}\ \mathsf{else}\ x_{p,n-1} \quad @\ R_{i+1}^\infty \\ \mapsto_{\mathsf{e}}^* & 1 + x_{p,n-1}\ \ @\ R_{i+1}^\infty\ \ \textit{if}\ b_p = \mathsf{True}; \\ & x_{p,n-1}\ \ @\ R_{i+1}^\infty\ \ \textit{otherwise}. \end{array}$$

By induction hypothesis, $\mathit{binomial}_p\ (n-1)$ generates a sample $x_{p,n-1}$ from $\mathit{Binomial}_{p,n-1}$ after consuming $R_1 \cdots R_{i-1}$ for some i (which is actually n). Since R_i is an independent random variable, $\mathit{bernoulli}_p$ generates a sample b_p that is independent of $x_{p,n-1}$. Then we obtain an outcome k with the probability of

$b_p = \mathsf{True}$ and $x_{p,n-1} = k-1$ or
$b_p = \mathsf{False}$ and $x_{p,n-1} = k$,
which is equal to $p * \mathit{Binomial}_{p,n-1}(k-1) + (1.0-p) * \mathit{Binomial}_{p,n-1}(k) = \mathit{Binomial}_{p,n}(k)$. Thus we have $E_{p,n} \sim \mathit{Binomial}_{p,n}$. $\square$

As a final example, we show that *geometric_rec p* denotes a geometric distribution with parameter p. Suppose $\mathit{geometric} \mapsto_{\mathsf{t}}^{*} \mathsf{prob}\ E$ and $E \sim \mathit{Prob}$. The computation of E proceeds as follows:

$$\begin{array}{lll}
 & E & @\ R_1^\infty \\
\mapsto_{\mathsf{e}}^{*} & \mathsf{sample}\ b\ \mathsf{from\ prob}\ b_p\ \mathsf{in} & \\
 & \mathsf{eif}\ b\ \mathsf{then}\ 0 & \\
 & \mathsf{else}\ \ \mathsf{sample}\ x\ \mathsf{from}\ \mathit{geometric}\ \mathsf{in} & @\ R_2^\infty \\
 & \qquad 1+x & \\
\mapsto_{\mathsf{e}}^{*} & 0 \qquad\qquad\qquad\qquad\qquad @\ R_2^\infty\ \ \mathit{if}\ \ b_p = \mathsf{True}; & \\
 & \mathsf{sample}\ x\ \mathsf{from\ prob}\ E\ \mathsf{in}\ 1+x\ \ @\ R_2^\infty\ \ \mathit{otherwise}. &
\end{array}$$

The first case happens with probability p and we get $\mathit{Prob}(0) = p$. In the second case, we compute the same expression E with R_2^∞. Since all random variables are independent, R_2^∞ can be thought of as a fresh sequence of random variables. Therefore the computation of E with R_2^∞ returns samples from the same probability distribution *Prob* and we get $\mathit{Prob}(1+k) = (1.0-p) * \mathit{Prob}(k)$. Solving the two equations, we get $\mathit{Prob}(k) = p * (1.0-p)^{k-1}$, which is the probability mass function for a geometric distribution with parameter p.

The above approach can be thought of as an adaption of the methodology established in simulation theory and demonstrates how to connect simulation theory and $\lambda_{\bigcirc}$. The proof of the correctness of a sampling method in simulation theory is easily transcribed into a proof similar to those shown in this section by interpreting random numbers in simulation theory as random variables in $\lambda_{\bigcirc}$. Thus $\lambda_{\bigcirc}$ serves as a programming language in which sampling methods developed in simulation theory can be not only formally expressed but also formally reasoned about. All this is possible in part because an expression computation in $\lambda_{\bigcirc}$ is provided with an infinite sequence of random numbers to consume, or equivalently, because of the use of generalized sampling functions as the mathematical basis.

Measure-theoretic approach

As the accepted mathematical basis of probability theory is measure theory [Rudin 1986], an alternative approach would be to develop a denotational semantics based on measure theory by translating expressions into a measure-theoretic structure. The denotational semantics would be useful in answering such questions as:

— Does every expression in $\lambda_{\bigcirc}$ result in a (partial or total) measurable sampling function? Or is it possible to write a pathological expression that corresponds to no measurable sampling function?

— Does every expression in $\lambda_{\bigcirc}$ specify a probability distribution? Or is it possible to write a pathological expression that specifies no probability distribution?

— Can we encode any probability distribution in $\lambda_{\bigcirc}$? If not, what kinds of probability distributions are impossible to encode in $\lambda_{\bigcirc}$?

Measure theory allows certain (but not all) sampling functions to specify probability distributions. Consider a sampling function f from $(0.0, 1.0]$ to $\mathcal{D}$. If f is a measurable function, then $\mathcal{D}$ is a measurable space and f determines a unique probability measure μ such that $\mu(S) = \nu(f^{-1}(S))$ where ν is Lebesgue measure over the unit interval. The intuition is that S, as an event, is assigned a probability equal to the size of its inverse image under f.

If we ignore fixed point constructs and disallow expressions of higher-order types, it is straightforward to translate expressions into probability measures, since probability measures form a monad [Giry 1981; Ramsey and Pfeffer 2002] and expressions already follow a monadic syntax. Let us write $[M]_{\mathsf{term}}$ for the denotation of term M. Then we can translate each expression E into a probability measure $[E]_{\mathsf{exp}}$ as follows:

— $[\mathsf{prob}\ E]_{\mathsf{term}} = [E]_{\mathsf{exp}}$.
— $[M]_{\mathsf{exp}}(S) = 1$ if $[M]_{\mathsf{term}}$ is in S.
  $[M]_{\mathsf{exp}}(S) = 0$ if $[M]_{\mathsf{term}}$ is not in S.
— $[\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E]_{\mathsf{exp}} = \int f d[M]_{\mathsf{term}}$ where a function f is defined as $f(x) = [E]_{\mathsf{exp}}$ and $\int f d[M]_{\mathsf{term}}$ is an integral of f over probability measure $[M]_{\mathsf{term}}$.
— $[\mathcal{S}]_{\mathsf{exp}}$ is Lebesgue measure over the unit interval $(0.0, 1.0]$.

Note that the translation does not immediately reveal the probability measure corresponding to a given expression because it returns a *formula* for the probability measure rather than the probability measure itself. Hence, in order to obtain the probability measure, we have to go through essentially the same analysis as in the previous approach based on the methodology developed in simulation theory. Ultimately we have to invert a sampling function specified by a given expression (because an event is assigned a probability proportional to the size of its inverse image under the sampling function), which may not be easy to do in a mechanical way in the presence of various operators.

Once we add fixed point constructs to $\lambda_{\bigcirc}$, expressions should be translated into a domain-theoretic structure because of recursive equations. Specifically a term $\mathsf{fix}\ x{:}\bigcirc A.\, M$ gives rise to a recursion equation on type $\bigcirc A$, and if a measure-theoretic structure is used for the denotation of terms of type $\bigcirc A$, it is difficult to solve the recursive equation; only with a domain-theoretic structure, the recursive equation can be given a theoretical treatment. The work by Jones [Jones 1990] shows how such a domain-theoretic structure can be constructed.

We close this section with a conjecture on the measurability of sampling functions specifiable in $\lambda_{\bigcirc}$. For the sake of simplicity, we consider expressions of type real only. For such an expression E, we define a sampling function f_E such that $f_E(\omega) = r$ if and only if $E\ @\ \omega \rightharpoondown r\ @\ \omega'$ for some sampling sequence ω'. Note that in general, f_E is a partial sampling function because the computation of E may not terminate. Our conjecture is that any such partial sampling function is measurable:

Conjecture 5.2. *If $\cdot \vdash E \div \mathsf{real}$, then the partial sampling function f_E is measurable. That is, for any measurable set S of real numbers, its inverse image $f_E^{-1}(S)$ is also measurable.*

Then the subprobability distribution induced by f_E can be regarded as the denotation of E. The key part of the proof will be to set up measurable structures for all valid types in $\lambda_\bigcirc$, including those for lambda abstractions and probability terms.

6. APPROXIMATE COMPUTATION IN $\lambda_\bigcirc$

In this section, we discuss approximate reasoning in $\lambda_\bigcirc$ by the Monte Carlo method [MacKay 1998]. It approximately answers a query on a probability distribution by generating a large number of samples and then analyzing them. For example, we can approximate $p_{Mary\_calls|John\_calls}$ in the belief network example in Section 4, which is equal to the proportion of True's among an infinite number of samples from $Q_{Mary\_calls|John\_calls}$, by generating a large number of samples and counting the number of True's. Although the Monte Carlo method gives only an approximate answer, its accuracy improves with the number of samples. Moreover it requires a description of a method of generating samples from a probability distribution and is therefore particularly suitable for $\lambda_\bigcirc$, in which an expression is such a description.

Below we use the Monte Carlo method to implement the expectation query and the Bayes operation. Both implementations are provided as primitive constructs of $\lambda_\bigcirc$. These primitive constructs are analogues of unsafePerformIO [Peyton Jones and Wadler 1993] and runST [Launchbury and Peyton Jones 1995] of Haskell in that they allow term evaluations ($\approx$ effect-free evaluations in Haskell) to initiate expression computations ($\approx$ effectful computations in Haskell).

6.1 Expectation query

Among common queries on probability distributions, the most important is the expectation query. The expectation of a function f with respect to a probability distribution P is the mean of f over P, which we write as $\int f dP$. Other queries may be derived as special cases of the expectation query. For example, the mean of a probability distribution over real numbers is the expectation of an identity function; the probability of an event *Event* under a probability distribution P is $\int I_{Event} dP$ where $I_{Event}(x)$ is 1 if x is in *Event* and 0 if not.

The Monte Carlo method states that we can approximate $\int f dP$ with a set of samples $V_1, \cdots, V_n$ from P:

$$\lim_{n\to\infty} \frac{f(V_1) + \cdots + f(V_n)}{n} = \int f dP$$

We introduce a term construct expectation which exploits the above equation:

$$\text{term } M ::= \cdots \mid \mathsf{expectation}\ M_f\ M_P$$

$$\frac{\Gamma \vdash M_f : A \to \mathsf{real} \quad \Gamma \vdash M_P : \bigcirc A}{\Gamma \vdash \mathsf{expectation}\ M_f\ M_P : \mathsf{real}}\ \mathsf{Exp}$$

$$\frac{\begin{array}{l} M_f \mapsto_{\mathsf{t}}^{*} f \quad M_P \mapsto_{\mathsf{t}}^{*} \mathsf{prob}\ E_P \\ \text{for } i = 1, \cdots, n \quad \text{new sampling sequence } \omega_i \quad E_P \mathbin{@} \omega_i \mapsto_{\mathsf{e}}^{*} V_i \mathbin{@} \omega_i' \quad f\ V_i \mapsto_{\mathsf{t}}^{*} v_i \end{array}}{\mathsf{expectation}\ M_f\ M_P \mapsto_{\mathsf{t}} \frac{\sum_i v_i}{n}}\ \mathit{Exp}$$

The rule *Exp* says that if M_f evaluates to a lambda abstraction denoting f and M_P evaluates to a probability term denoting P, then expectation M_f M_P reduces to an approximation of $\int f dP$. A run-time variable n, which can be chosen by programmers for each expectation query in the source program, specifies the number of samples to generate from P. To evaluate $\mathsf{expectation}\ M_f\ M_P$, the run-time system initializes sampling sequence ω_i to generate sample V_i for $i = 1, \cdots, n$ (as indicated by new sampling sequence ω_i).

In the rule *Exp*, the convergence of samples V_i with respect to probability distribution P and the accuracy of $\frac{\sum_i v_i}{n}$ are controlled not by $\lambda_\bigcirc$ but solely by programmers. That is, $\lambda_\bigcirc$ is not responsible for choosing a value of n (*e.g.*, by analyzing E_P) to guarantee a certain level of accuracy in estimating $\int f dP$. Rather it is programmers that decide an expression E_P for encoding P as well as a suitable value of n to achieve a desired level of accuracy. We do not consider this as a weakness of $\lambda_\bigcirc$, since E_P itself (chosen by programmers) affects the accuracy of $\frac{\sum_i v_i}{n}$ after all.

Although $\lambda_\bigcirc$ provides no concrete guidance in choosing a value of n in the rule *Exp*, programmers can analytically or empirically determine, for each expectation query, a suitable value of n that finishes the expectation query within a given time constraint. (In general, a large value of n is better because it results in a more faithful approximation of P by samples V_i and a smaller difference between $\frac{\sum_i v_i}{n}$ and the true expectation $\int f dP$.) Ideally the amount of time required to evaluate $\mathsf{expectation}\ M_f\ M_P$ should be directly proportional to n, but in practice, the computation of the same expression E_P may take a different amount of time, especially if E_P expresses a recursive computation. Therefore programmers can try different values of n to find the largest one that finishes the expectation query within a given time constraint.

Now we can calculate $p_{Mary\_calls|John\_calls}$ from Section 4 as follows:

$$\mathsf{expectation}\ (\lambda x\!:\!\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ 1.0\ \mathsf{else}\ 0.0)\ Q_{Mary\_calls|John\_calls}$$

If we use parameters given on page 439 of [Russell and Norvig 1995] with $P_{alarm|burglary}$ set to *bernoulli* 0.95, the closed-form solution of $p_{Mary\_calls|John\_calls}$ yields 0.0400859927044. The following table shows approximate values of $p_{Mary\_calls|John\_calls}$ as the number of samples increases:

number of samples	$p_{Mary\_calls\|John\_calls}$
100	0.04
1000	0.041
10000	0.0416
100000	0.04012
1000000	0.040064

Note that the accuracy in approximating $p_{Mary\_calls|John\_calls}$ increases with the number of samples. See Section 6.4 for the time required to generate samples.

6.2 Bayes operation

The previous implementation of the Bayes operation $P \sharp Q$ assumes a function p and a constant c such that $p(x) = kP(x) \leq c$ for a certain constant k. It is, however, often difficult to find the optimal value of c (*i.e.*, the maximum value of $p(x)$) and we have to take a conservative estimate of c. The Monte Carlo method, in conjunction with importance sampling [MacKay 1998], allows us to dispense with

c by approximating Q with a set of samples and $P \sharp Q$ with a set of weighted samples. We introduce a term construct bayes for the Bayes operation and an expression construct importance for importance sampling:

$$\begin{array}{lll} \text{term} & M & ::= \cdots \mid \mathsf{bayes}\ M_p\ M_Q \\ \text{expression} & E & ::= \cdots \mid \mathsf{importance}\ \{(V_i, w_i) | 1 \le i \le n\} \end{array}$$

In the spirit of data abstraction, importance represents only an internal data structure and is not directly available to programmers.

$$\frac{\Gamma \vdash M_p : A \rightarrow \mathsf{real} \quad \Gamma \vdash M_Q : \bigcirc A}{\Gamma \vdash \mathsf{bayes}\ M_p\ M_Q : \bigcirc A}\ \mathsf{Bayes}$$

$$\frac{\Gamma \vdash V_i : A \quad \Gamma \vdash w_i : \mathsf{real} \quad 1 \le i \le n}{\Gamma \vdash \mathsf{importance}\ \{(V_i, w_i) | 1 \le i \le n\} \div A}\ \mathsf{Imp}$$

$$\frac{\begin{array}{l} M_p \mapsto_{\mathsf{t}}^{*} p \quad M_Q \mapsto_{\mathsf{t}}^{*} \mathsf{prob}\ E_Q \\ \text{for } i = 1, \cdots, n \quad \text{new sampling sequence } \omega_i \quad E_Q\ @\ \omega_i \mapsto_{\mathsf{e}}^{*} V_i\ @\ \omega_i' \quad p\ V_i \mapsto_{\mathsf{t}}^{*} w_i \end{array}}{\mathsf{bayes}\ M_p\ M_Q \mapsto_{\mathsf{t}} \mathsf{prob}\ \mathsf{importance}\ \{(V_i, w_i) | 1 \le i \le n\}}\ \mathit{Bayes}$$

$$\frac{\frac{\sum_{i=1}^{k-1} w_i}{S} < r \le \frac{\sum_{i=1}^{k} w_i}{S} \quad \mathit{where} \quad S = \sum_{i=1}^{n} w_i}{\mathsf{importance}\ \{(V_i, w_i) | 1 \le i \le n\}\ @\ r\omega \mapsto_{\mathsf{e}} V_k\ @\ \omega}\ \mathit{Imp}$$

The rule *Bayes* uses sampling sequences $\omega_1, \cdots, \omega_n$ initialized by the run-time system and approximates Q with n samples $V_1, \cdots, V_n$, where n is a run-time variable as in the rule *Exp*. Then it applies p to each sample V_i to calculates its weight w_i and creates a set $\{(V_i, w_i) | 1 \le i \le n\}$ of weighted samples as an argument to importance. The rule *Imp* implements importance sampling: we use a random number r to probabilistically select a sample V_k by taking into account the weights associated with all the samples.

6.3 expectation and bayes as term constructs

A problem with the above definition is that although expectation and Bayes are term constructs, their reduction is probabilistic because of sampling sequence ω_i in the rules *Exp* and *Bayes*. This violates the principle that a term evaluation is always deterministic, and now the same term may evaluate to different values. For pragmatic reasons, however, we still choose to define expectation and bayes as term constructs rather than expression constructs. Consider a probability distribution $P(s)$ defined in terms of probability distributions $Q(s)$ and $R(u)$:

$$P(s) = \eta Q(s) \int f(s, u) R(u) du$$

(A similar example is found in Section 8.3.) $P(s)$ is obtained by the Bayes operation between $Q(s)$ and $p(s) = \int f(s, u) R(u) du$, and is encoded in $\lambda_{\bigcirc}$ as

$$\mathsf{bayes}\ (\lambda s\!:\!\_.\ \mathsf{expectation}\ (\lambda u\!:\!\_.\ M_f\ (s, u))\ M_R)\ M_Q$$

where M_R and M_Q are probability terms denoting R and Q, respectively, and M_f is a lambda abstraction denoting f. If expectation was an expression construct, however, it would be difficult to encode $P(s)$ because expression expectation $(\lambda u\!:\!\_.\ M_f(s, u))\ M_Q$

cannot be converted into a term. In essence, mathematically the expectation of a function with respect to a probability distribution and the result of a Bayes operation are always unique (if they exist), which in turn implies that if expectation and bayes are defined as expression constructs, we cannot write code involving expectations and Bayes operations in the same manner that we reason mathematically.

In the actual implementation of $\lambda_{\bigcirc}$ (to be presented in the next section), terms are not protected from computational effects (such as input/output and mutable references) and term evaluations do not always result in unique values anyway. Hence non-deterministic term evaluations should not be regarded as a new problem, and expressions are best interpreted as a syntactic category dedicated to probabilistic computations only in the mathematical sense. Strict adherence at the implementation level to the semantic distinction between terms and expressions (*e.g.*, defining expectation and bayes as expression constructs) would cost code readability without any apparent benefit.

6.4 Cost of generating random numbers

The essence of the Monte Carlo method is to trade accuracy for cost — it only gives approximate answers, but relieves programmers of the cost of exact computation (which can be even impossible in certain problems). Since $\lambda_{\bigcirc}$ relies on the Monte Carlo method to reason about probability distributions, it is important for programmers to be able to determine the cost of the Monte Carlo method.

We decide to define the cost of the Monte Carlo method as proportional to the number of random numbers consumed. The decision is based on the assumption that random number generation can account for a significant portion of the total computation time. Under our implementation of $\lambda_{\bigcirc}$, random number generation for the following examples from Section 4 accounts for an average of 74.85% of the total computation time. The following table shows execution times in seconds and percentages of random number generation when generating 100,000 samples (on a Pentium III 500Mhz with 384 MBytes memory):

test case	execution time	random number generation (%)
uniform 0.0 1.0	0.25	78.57
binomial 0.25 16	4.65	64.84
geometric_efix 0.25	1.21	63.16
gaussian_rejection 2.5 5.0	1.13	77.78
$exponential\_von\_Neumann_{1.0}$	1.09	80.76
gaussian_Box_Muller 2.0 4.0	0.57	77.27
gaussian_central 0.0 1.0	2.79	83.87
$Q_{Mary\_calls \mid John\_calls}$	21.35	72.57

In $\lambda_{\bigcirc}$, it is the programmers' responsibility to reason about the cost of generating random numbers, since for an expression computation judgment $E\ @\ \omega \rightharpoondown V\ @\ \omega'$, the length of the consumed sequence $\omega - \omega'$ is not observable. A analysis similar to those in Section 5 can be used to estimate the cost of obtaining a sample in terms of the number of random numbers consumed. In the case of *geometric_rec* p, for example, the expected number n of random numbers consumed is calculated by solving the equation

$$n = 1 + (1 - p) * n$$

where 1 accounts for the random number generated from the Bernoulli distribution and $(1 - p)$ is the probability that another attempt is made to generate a sample from the same probability distribution. The same technique applies equally to the rejection method (*e.g.*, *gaussian_rejection*). In general, analyses in simulation theory for calculating the expected number of random numbers consumed can be transcribed to similar analyses for $\lambda_{\bigcirc}$ in a straightforward way, since $\lambda_{\bigcirc}$, as a programming language with formal semantics, allows us to express sampling methods developed in simulation theory.

7. IMPLEMENTATION

This section describes the implementation of $\lambda_{\bigcirc}$. Instead of implementing $\lambda_{\bigcirc}$ as a complete programming language of its own, we choose to embed it in an existing functional language for two pragmatic reasons. First the conceptual basis of probabilistic computations in $\lambda_{\bigcirc}$ is simple enough that it is easy to simulate all language constructs of $\lambda_{\bigcirc}$ without any modification to the run-time system. Second we intend to use $\lambda_{\bigcirc}$ for real applications in robotics, for which we wish to exploit advanced features such as a module system, an interface to foreign languages, and a graphics library. Hence building a complete compiler for $\lambda_{\bigcirc}$ is not justified when extending an existing functional language is sufficient for examining the practicality of $\lambda_{\bigcirc}$.

In our implementation, we use Objective CAML as the host language. Since such constructs of $\lambda_{\bigcirc}$ as probability terms and bind expressions are not available in Objective CAML, we first extend the syntax of Objective CAML using CAMLP4, a preprocessor for Objective CAML, so as to incorporate the syntax of $\lambda_{\bigcirc}$. The extended syntax is then translated back in the original syntax of Objective CAML.

For the sake of simplicity, we assume that $\lambda_{\bigcirc}$ uses floating point numbers in place of real numbers, since the overhead of exact real arithmetic is not justified in $\lambda_{\bigcirc}$ where we work with samples and approximations. We formulate a sound and complete translation of $\lambda_{\bigcirc}$ in a simple call-by-value language which can be thought of a sublanguage of Objective CAML.

7.1 Representation of sampling functions

Since a probability term denotes a probability distribution specified by a sampling function, the implementation of $\lambda_{\bigcirc}$ translates probability terms into representations of sampling functions. We translate a probability term of type $\bigcirc A$ into a value of type A `prob`, where the type constructor `prob` is conceptually defined as follows:

$$\texttt{type}\ A\ \texttt{prob}\ =\ \texttt{float}^{\infty} \texttt{->} A * \texttt{float}^{\infty}$$

`float` is the type of floating point numbers, and we use $\texttt{float}^{\infty}$ for the type of infinite sequences of random numbers. The actual definition of `prob` dispenses with infinite sequences of random numbers by using a global random number generator whenever fresh random numbers are needed to compute sampling expressions:

$$\texttt{type}\ A\ \texttt{prob}\ =\ \texttt{unit} \texttt{->} A$$

type	A, B	$::=$	$A \rightarrow A \mid \bigcirc A \mid \mathsf{real}$
term	M, N	$::=$	$x \mid \lambda x\!:\!A.\, M \mid M\ M \mid \mathsf{prob}\ E \mid r$
expression	E, F	$::=$	$M \mid \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E \mid \mathcal{S} \mid \mathbf{x} \mid \mathsf{efix}\ \mathbf{x}\div A.\, E$
value/sample	V	$::=$	$\lambda x\!:\!A.\, M \mid \mathsf{prob}\ E \mid r$
floating point number	r		
sampling sequence	ω	$::=$	$r_1 r_2 \cdots r_i \cdots$ *where* $r_i \in (0.0, 1.0]$
typing context	Γ	$::=$	$\cdot \mid \Gamma, x : A \mid \Gamma, \mathbf{x} \div A$

Fig. 4. A fragment of $\lambda_\bigcirc$ as the source language.

type	A, B	$::=$	$A \texttt{->} A \mid A\ \texttt{prob} \mid \texttt{float} \mid \texttt{unit}$
expression	e, f	$::=$	$x \mid \texttt{fun}\ x\!:\!A.\ e \mid e\ e \mid \texttt{prb}\ e \mid \texttt{app}\ e \mid r \mid$
			$() \mid \texttt{random} \mid \texttt{fix}\ x\!:\!A.\ u$
value	v	$::=$	$\texttt{fun}\ x\!:\!A.\ e \mid \texttt{prb}\ v \mid r \mid ()$
function	u	$::=$	$\texttt{fun}\ x\!:\!A.\ e$
floating point number	r		
sampling sequence	ω	$::=$	$r_1 r_2 \cdots r_i \cdots$ *where* $r_i \in (0.0, 1.0]$
typing context	Γ	$::=$	$\cdot \mid \Gamma, x : A$

Fig. 5. A call-by-value language as the target language.

Here `unit` is the unit type which is inhabited only by a unit value ().

We use the type constructor `prob` as an abstract datatype. That is, the definition of `prob` is not exposed to $\lambda_\bigcirc$ and values of type A `prob` are accessed only via member functions. We provide two member functions: `prb` and `app`. `prb` builds a value of type A `prob` from a function of type `unit` −> A; it is actually defined as an identity function. `app` generates a sample from a value of type A `prob`; it applies its argument to a unit value. The interface and implementation of the abstract datatype `prob` are given as follows:

```
type A prob                           type A prob  =  unit -> A
val prb  : (unit -> A) -> A prob      let prb  = fun f:unit -> A. f
val app  : A prob -> A                let app  = fun f:A prob. f ()
```

We use `prb` in translating probability terms and `app` in translating bind expressions. In conjunction with the use of the type constructor `prob` as an abstract data type, they provide a sound and complete translation of $\lambda_\bigcirc$, as shown in the next subsection.

7.2 Translation of $\lambda_\bigcirc$ in a call-by-value language

We translate a fragment of $\lambda_\bigcirc$ shown in Figure 4 in a call-by-value language shown in Figure 5. The source language excludes product types, which are straightforward to translate if the target language is extended with product types. We directly translate expression fixed point constructs without simulating them with fixed point constructs for terms. As the target language supports only floating point numbers, r in the source language is restricted to floating point numbers.

The target language is a call-by-value language extended with the abstract datatype `prob`. It has a single syntactic category consisting of expressions (because it does not distinguish between effect-free evaluations and effectful computations). As in

$$\frac{}{\Gamma, x : A \vdash_{\mathtt{v}} x : A}\ \mathtt{Hyp} \qquad \frac{\Gamma, x : A \vdash_{\mathtt{v}} e : B}{\Gamma \vdash_{\mathtt{v}} \mathtt{fun}\ x{:}A.\ e : A \mathtt{\ ->\ } B}\ \mathtt{Lam} \qquad \frac{\Gamma \vdash_{\mathtt{v}} e_1 : A \mathtt{\ ->\ } B \quad \Gamma \vdash_{\mathtt{v}} e_2 : A}{\Gamma \vdash_{\mathtt{v}} e_1\ e_2 : B}\ \mathtt{App}$$

$$\frac{\Gamma \vdash_{\mathtt{v}} e : \mathtt{unit} \mathtt{\ ->\ } A}{\Gamma \vdash_{\mathtt{v}} \mathtt{prb}\ e : A\ \mathtt{prob}}\ \mathtt{Prb} \qquad \frac{\Gamma \vdash_{\mathtt{v}} e : A\ \mathtt{prob}}{\Gamma \vdash_{\mathtt{v}} \mathtt{app}\ e : A}\ \mathtt{Papp} \qquad \frac{}{\Gamma \vdash_{\mathtt{v}} r : \mathtt{float}}\ \mathtt{Float}$$

$$\frac{}{\Gamma \vdash_{\mathtt{v}} () : \mathtt{unit}}\ \mathtt{Unit} \qquad \frac{}{\Gamma \vdash_{\mathtt{v}} \mathtt{random} : \mathtt{float}}\ \mathtt{Random} \qquad \frac{\Gamma, x : A \vdash_{\mathtt{v}} u : A}{\Gamma \vdash_{\mathtt{v}} \mathtt{fix}\ x{:}A.\ u : A}\ \mathtt{Fix}$$

Fig. 6. Typing rules of the target language.

$$\frac{e \ @\ \omega \mapsto_{\mathtt{v}} e' \ @\ \omega'}{e\ f \ @\ \omega \mapsto_{\mathtt{v}} e'\ f \ @\ \omega'}\ \mathtt{E}_{\beta_{\mathtt{L}}} \qquad \frac{f \ @\ \omega \mapsto_{\mathtt{v}} f' \ @\ \omega'}{(\mathtt{fun}\ x{:}A.\ e)\ f \ @\ \omega \mapsto_{\mathtt{v}} (\mathtt{fun}\ x{:}A.\ e)\ f' \ @\ \omega'}\ \mathtt{E}_{\beta_{\mathtt{R}}}$$

$$\frac{}{(\mathtt{fun}\ x{:}A.\ e)\ v \ @\ \omega \mapsto_{\mathtt{v}} [v/x]e \ @\ \omega}\ \mathtt{E}_{\beta_{\mathtt{V}}} \qquad \frac{e \ @\ \omega \mapsto_{\mathtt{v}} e' \ @\ \omega'}{\mathtt{prb}\ e \ @\ \omega \mapsto_{\mathtt{v}} \mathtt{prb}\ e' \ @\ \omega'}\ \mathtt{E}_{\mathtt{Prb}}$$

$$\frac{e \ @\ \omega \mapsto_{\mathtt{v}} e' \ @\ \omega'}{\mathtt{app}\ e \ @\ \omega \mapsto_{\mathtt{v}} \mathtt{app}\ e' \ @\ \omega'}\ \mathtt{E}_{\mathtt{App}} \qquad \frac{}{\mathtt{app}\ \mathtt{prb}\ v \ @\ \omega \mapsto_{\mathtt{v}} v\ () \ @\ \omega}\ \mathtt{E}_{\mathtt{AppPrb}}$$

$$\frac{}{\mathtt{random} \ @\ r\omega \mapsto_{\mathtt{v}} r \ @\ \omega}\ \mathtt{E}_{\mathtt{Random}} \qquad \frac{}{\mathtt{fix}\ x{:}A.\ u \ @\ \omega \mapsto_{\mathtt{v}} [\mathtt{fix}\ x{:}A.\ u/x]u \ @\ \omega}\ \mathtt{E}_{\mathtt{Fix}}$$

Fig. 7. Operational semantics of the target language.

$\lambda_{\bigcirc}$, every expression denotes a probabilistic computation and we say that an expression computes to a value. Note that fixed point constructs $\mathtt{fix}\ x{:}A.\ u$ allow recursive expressions only over function types.

The type system of the target language is shown in Figure 6. It employs a typing judgment $\Gamma \vdash_{\mathtt{v}} e : A$, meaning that expression e has type A under typing context Γ. The rules Prb and Papp conform to the interface of the abstract datatype prob.

The operational semantics of the target language is shown in Figure 7. It employs an expression reduction judgment $e \ @\ \omega \mapsto_{\mathtt{v}} e' \ @\ \omega'$, meaning that the computation of e with sampling sequence ω reduces to the computation of e' with sampling sequence ω'. A capture-avoiding substitution $[e/x]f$ is defined in a standard way. The rule $\mathtt{E}_{\mathtt{AppPrb}}$ is defined according to the implementation of the abstract datatype prob. The rule $\mathtt{E}_{\mathtt{Random}}$ shows that random, like sampling expressions in $\lambda_{\bigcirc}$, consumes a random number in a given sampling sequence. We write $\mapsto_{\mathtt{v}}^{*}$ for the reflexive and transitive closure of $\mapsto_{\mathtt{v}}$.

Figure 8 shows the translation of the source language in the target language.[3] We overload the function $[\cdot]_{\mathtt{v}}$ for types, typing contexts, terms, and expressions. Both terms and expressions of type A in the source language are translated into expressions of type $[A]_{\mathtt{v}}$ in the target language. $[\mathsf{prob}\ E]_{\mathtt{v}}$ suspends the computation of $[E]_{\mathtt{v}}$ by building a function $\mathtt{fun}\ \_{:}\mathtt{unit}.\ [E]_{\mathtt{v}}$, just as $\mathsf{prob}\ E$ suspends the computation of E. Since the target language allows recursive expressions only over function types, an expression variable $\mathbf{x}$ of type A (*i.e.*, $\mathbf{x} \div A$) is translated into $x_{\mathbf{x}}\ ()$ where $x_{\mathbf{x}}$ is a special variable of type $\mathtt{unit} \mathtt{\ ->\ } [A]_{\mathtt{v}}$ annotated with $\mathbf{x}$; if the target language allowed recursive expressions over any type, $\mathbf{x}$ and $\mathsf{efix}\ \mathbf{x}{\div}A.\,E$

[3] _ is a wildcard pattern for variables and types.

$$
\begin{aligned}
[A \to B]_{\mathtt{v}} &= [A]_{\mathtt{v}} \;\texttt{->}\; [B]_{\mathtt{v}} \\
[\bigcirc A]_{\mathtt{v}} &= [A]_{\mathtt{v}} \;\texttt{prob} \\
[\mathsf{real}]_{\mathtt{v}} &= \texttt{float} \\
\\
[\cdot]_{\mathtt{v}} &= \cdot \\
[\Gamma, x : A]_{\mathtt{v}} &= [\Gamma]_{\mathtt{v}}, x : [A]_{\mathtt{v}} \\
[\Gamma, \mathbf{x} \div A]_{\mathtt{v}} &= [\Gamma]_{\mathtt{v}}, x_{\mathbf{x}} : \texttt{unit} \;\texttt{->}\; [A]_{\mathtt{v}} \\
\\
[x]_{\mathtt{v}} &= x \\
[\lambda x \!:\! A.\, M]_{\mathtt{v}} &= \texttt{fun}\; x \!:\! [A]_{\mathtt{v}}.\; [M]_{\mathtt{v}} \\
[M\ N]_{\mathtt{v}} &= [M]_{\mathtt{v}}\ [N]_{\mathtt{v}} \\
[\mathsf{prob}\ E]_{\mathtt{v}} &= \texttt{prb}\ (\texttt{fun}\ \_ \!:\! \texttt{unit}.\ [E]_{\mathtt{v}}) \\
[r]_{\mathtt{v}} &= r \\
[\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E]_{\mathtt{v}} &= (\texttt{fun}\ x \!:\! \_.\ [E]_{\mathtt{v}})\ (\texttt{app}\ [M]_{\mathtt{v}}) \\
[\mathcal{S}]_{\mathtt{v}} &= \texttt{random} \\
[\mathbf{x}]_{\mathtt{v}} &= x_{\mathbf{x}}\ () \\
[\mathsf{efix}\ \mathbf{x} \div A.\, E]_{\mathtt{v}} &= (\texttt{fix}\ x_{\mathbf{x}} \!:\! \texttt{unit} \;\texttt{->}\; [A]_{\mathtt{v}}.\ \texttt{fun}\ \_ \!:\! \texttt{unit}.\ [E]_{\mathtt{v}})\ ()
\end{aligned}
$$

Fig. 8. Translation of the source language.

could be translated into $x_{\mathbf{x}}$ and $\texttt{fix}\ x_{\mathbf{x}} \!:\! [A]_{\mathtt{v}}.\ [E]_{\mathtt{v}}$, respectively.[4]

Propositions 7.1 and 7.2 show that the translation is faithful to the type system of the source language. Proposition 7.1 proves the soundness of the translation: a well-typed term or expression in the source language is translated into a well-typed expression in the target language. Proposition 7.2 proves the completeness of the translation: only a well-typed term or expression in the source language is translated into a well-type expression in the target language. See Appendix A.3 for a proof.

PROPOSITION 7.1.
If $\Gamma \vdash M : A$, *then* $[\Gamma]_{\mathtt{v}} \vdash_{\mathtt{v}} [M]_{\mathtt{v}} : [A]_{\mathtt{v}}$.
If $\Gamma \vdash E \div A$, *then* $[\Gamma]_{\mathtt{v}} \vdash_{\mathtt{v}} [E]_{\mathtt{v}} : [A]_{\mathtt{v}}$.

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

PROPOSITION 7.2.
If $[\Gamma]_{\mathtt{v}} \vdash_{\mathtt{v}} [M]_{\mathtt{v}} : A$, *then there exists* B *such that* $A = [B]_{\mathtt{v}}$ *and* $\Gamma \vdash M : B$.
If $[\Gamma]_{\mathtt{v}} \vdash_{\mathtt{v}} [E]_{\mathtt{v}} : A$, *then there exists* B *such that* $A = [B]_{\mathtt{v}}$ *and* $\Gamma \vdash E \div B$.

The translation is also faithful to the operational semantics of the source language. We first show that the translation is sound: a term reduction in the source language is translated into a corresponding expression reduction which consumes no random number (Proposition 7.3); an expression reduction in the source language is translated into a corresponding sequence of expression reductions which consumes the same sequence of random numbers (Proposition 7.4). Note that in Proposition 7.4, $[E]_{\mathtt{v}}$ does not reduce to $[F]_{\mathtt{v}}$; instead it reduces to an expression e to which $[F]_{\mathtt{v}}$ eventually reduces without consuming random numbers. See Appendix A.4 for proofs.

[4]In the Objective CAML syntax, $[\mathsf{efix}\ \mathbf{x} \div A.\, E]_{\mathtt{v}}$ can be rewritten as `let rec` $x_{\mathbf{x}}$ `() =` $[E]_{\mathtt{v}}$ `in` $x_{\mathbf{x}}$ `()`.

Proposition 7.3.
If $M \mapsto_{\mathsf{t}} N$, then $[M]_{\mathsf{v}} @ \omega \mapsto_{\mathsf{v}} [N]_{\mathsf{v}} @ \omega$ for any sampling sequence ω.

Proposition 7.4.
If $E @ \omega \mapsto_{\mathsf{e}} F @ \omega'$, there exists e such that $[E]_{\mathsf{v}} @ \omega \mapsto_{\mathsf{v}}^{*} e @ \omega'$ and $[F]_{\mathsf{v}} @ \omega' \mapsto_{\mathsf{v}}^{*} e @ \omega'$.

The completeness of the translation states that only a valid term or expression reduction in the source language is translated into a corresponding sequence of expression reductions in the target language. In other words, a term or expression that cannot be further reduced in the source language is translated into an expression whose reduction eventually gets stuck. To simplify the presentation, we introduce three judgments, all of which express that a term or expression does not further reduces.

— $M \mapsto_{\mathsf{t}} \bullet$ means that there exists no term to which M reduces.
— $E @ \omega \mapsto_{\mathsf{e}} \bullet$ means that there exists no expression to which E reduces.
— $e @ \omega \mapsto_{\mathsf{v}} \bullet$ means that there exists no expression to which e reduces (in the target language).

Corollary 7.6 proves the completeness of the translation for terms; Proposition 7.7 proves the completeness of the translation for expressions. See Appendix A.5 for proofs.

Proposition 7.5. *If $[M]_{\mathsf{v}} @ \omega \mapsto_{\mathsf{v}} e @ \omega'$, then $e = [N]_{\mathsf{v}}$, $\omega = \omega'$, and $M \mapsto_{\mathsf{t}} N$.*

Corollary 7.6. *If $M \mapsto_{\mathsf{t}} \bullet$, then $[M]_{\mathsf{v}} @ \omega \mapsto_{\mathsf{v}} \bullet$ for any sampling sequence ω.*

Proposition 7.7.
If $E @ \omega \mapsto_{\mathsf{e}} \bullet$, then there exists e such that $[E]_{\mathsf{v}} @ \omega \mapsto_{\mathsf{v}}^{*} e @ \omega \mapsto_{\mathsf{v}} \bullet$.

The target language can be thought of as a sublanguage of Objective CAML in which the abstract datatype `prob` is built-in and `random` is implemented as `Random.float 1.0`.[5] A sampling sequence ω in an expression reduction judgment can be thought of as part of the state of the run-time system, namely the state of the random number generator. Since Objective CAML also serves as the host language for $\lambda_{\bigcirc}$, we extend the syntax of Objective CAML to incorporate the syntax of $\lambda_{\bigcirc}$ (using CAMLP4). The extended syntax is then translated back in the original syntax of Objective CAML using the function $[\cdot]_{\mathsf{v}}$.

7.3 Discussion

Although $\lambda_{\bigcirc}$ is implemented indirectly via a translation in Objective CAML, both its type system and its operational semantics are faithfully mirrored through the use of an abstract datatype. Besides all existing features of Objective CAML are available when programming in $\lambda_{\bigcirc}$, and we may think of the implementation of $\lambda_{\bigcirc}$ as a conservative extension of Objective CAML. The translation is easily generalized to any monadic language, thus complementing the well-established result that

[5]To be strict, `random` would be implemented as `1.0 -. Random.float 1.0`.

a call-by-value language is translated in a monadic language (*e.g.*, see [Sabry and Wadler 1997]).

The translator of $\lambda_{\bigcirc}$ does not protect terms from computational effects already available in Objective CAML such as input/output, mutable references, and even direct uses of `Random.float`. Thus, for example, term M in a bind expression $\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E$ is supposed to produce no computational effect, but the translator has no way to verify that the evaluation of M is effect-free. Therefore the translator of $\lambda_{\bigcirc}$ relies on programmers to ensure that every term denotes a regular value.

We could directly implement $\lambda_{\bigcirc}$ by extending the compiler and the run-time system of Objective CAML. An immediate benefit is that type error messages are more informative because type errors are detected at the level of $\lambda_{\bigcirc}$. (Our implementation detects type errors in the translated code rather than in the source code; hence programmers should analyze type error messages to locate type errors in the source code.) As for execution speed, we conjecture that the gain is negligible, since the only overhead incurred by the abstract datatype `prob` is to invoke two tiny functions when its member functions are invoked: an identity function (for `prb`) and a function applying its argument to a unit value (for `app`).

8. APPLICATIONS

This section presents three applications of $\lambda_{\bigcirc}$ in robotics: robot localization, people tracking, and robotic mapping, all of which are popular topics in robotics. Although different in goal, all these applications share a common characteristic: the state of a robot is estimated from sensor readings, where the definition of state differs in each case. A key element of these applications is uncertainty in sensor readings, due to limitations of sensors and noise from the environment. It makes the problem of estimating the state of a robot both interesting and challenging: if all sensor readings were accurate, the state of a robot could be accurately traced by a simple (non-probabilistic) analysis of sensor readings. In order to cope with uncertainty in sensor readings, we estimate the state of a robot with probability distributions.

As a computational framework, we use Bayes filters. In each case, we formulate the update equations at the level of probability distributions and translate them in $\lambda_{\bigcirc}$. All implementations are tested using data collected with real robots. Experimental results are found at `http://www.cs.cmu.edu/~gla/toplas05/`.

8.1 Sensor readings: action and measurement

To update the state of a robot, we use two kinds of sensor readings: *action* and *measurement*. As in a Bayes filter, an action induces a state change whereas a measurement gives information on the state:

— An action a is represented as an odometer reading which returns the pose (*i.e.*, position (x, y) and orientation θ) of the robot relative to its initial pose. It is given as a tuple $(\Delta x, \Delta y, \Delta\theta)$.

— A measurement m consists of range readings which return distances to objects visible at certain angles. It is given as an array $[d_1; \cdots; d_n]$ where each d_i, $1 \leq i \leq n$, denotes the distance between the robot and the closest object visible at a certain angle.

Odometers and range finders are prone to errors because of their mechanical nature. An odometer usually tends to drift in one direction over time. Its accumulated error becomes manifest especially when the robot closes a loop after taking a circular route. Range finders occasionally fail to recognize obstacles and report the maximum distance measurable. In order to correct these errors, we use a probabilistic approach by representing the state of the robot with a probability distribution.

In the probabilistic approach, an action increases the set of possible states of the robot because it induces a state change probabilistically. In contrast, a measurement decreases the set of possible states of the robot because it gives negative information on unlikely states (and positive information on likely states). We now demonstrate how to probabilistically update the state of the robot in three different applications.

8.2 Robot localization

Robot localization [Thrun 2000a] is the problem of estimating the pose of a robot when a map of the environment is available. If the initial pose is given, the problem becomes *pose tracking* which keeps track of the robot pose by compensating errors in sensor readings. If the initial pose is not given, the problem becomes *global localization* which begins with multiple hypotheses on the robot pose (and is therefore more involved than pose tracking).

We consider robot localization under the assumption (called the *Markov assumption*) that the past and the future are independent if the current pose is known, or equivalently that the environment is static. This assumption allows us to use a Bayes filter in estimating the robot pose. Specifically the state in the Bayes filter is the robot pose $s = (x, y, \theta)$, and we estimate s with a probability distribution $Bel(s)$ over three-dimensional real space. We compute $Bel(s)$ according to the following update equations (which are the same as shown in Section 2):

$$Bel(s) \leftarrow \int \mathcal{A}(s|a, s') Bel(s') ds' \tag{3}$$

$$Bel(s) \leftarrow \eta \mathcal{P}(m|s) Bel(s) \tag{4}$$

η a normalizing constant ensuring $\int Bel(s)ds = 1.0$. We use the following interpretation of $\mathcal{A}(s|a, s')$ and $\mathcal{P}(m|s)$:

— $\mathcal{A}(s|a, s')$ is the probability that the robot moves to pose s after taking action a at another pose s'. $\mathcal{A}$ is called an *action model*.

— $\mathcal{P}(m|s)$ is the probability that measurement m is taken at pose s. $\mathcal{P}$ is called a *perception model*.

Given an action a and a pose s', we can generate a new pose s from $\mathcal{A}(\cdot|a, s')$ by adding a noise to a and applying it to s'. Given a measurement m and a pose s, we can also compute $\kappa\mathcal{P}(m|s)$ where κ is an unknown constant: the map determines a unique (accurate) measurement m_s for pose s, and the squared distance between m and m_s is assumed to be proportional to $\mathcal{P}(m|s)$. Then, if $M_{\mathcal{A}}$ denotes conditional probability $\mathcal{A}$ and $M_{\mathcal{P}}\ m$ returns a function $f(s) = \kappa\mathcal{P}(m|s)$, we implement update equations (3) and (4) as follows:

$$\left.\begin{array}{ll}\text{let } Bel_{new} = \textsf{prob} & \textsf{sample } s' \textsf{ from } Bel \textsf{ in} \\ & \textsf{sample } s \textsf{ from } M_{\mathcal{A}}\ (a, s') \textsf{ in} \\ & s\end{array}\right\} (3)$$

$$\text{let } Bel_{new} = \textsf{bayes}\ (M_{\mathcal{P}}\ m)\ Bel \quad \} \ (4)$$

Both pose tracking and global localization are achieved by specifying an appropriate initial probability distribution of robot pose. For pose tracking, we use a point-mass distribution or a Gaussian distribution; for global localization, we use a uniform distribution over the open space in the map.

8.3 People tracking

People tracking [Montemerlo et al. 2002] is an extension of robot localization in that it estimates not only the robot pose but also positions of people (or unmapped objects). As in robot localization, the robot takes an action to change its pose. Unlike in robot localization, however, the robot categorizes sensor readings in a measurement by deciding whether they correspond with objects in the map or with people. Those sensor readings that correspond with objects in the map are used to update the robot pose; the rest of sensor readings are used to update positions of people.

A simple approach is to maintain a probability distribution $Bel(s, \vec{u})$ of robot pose s and positions $\vec{u}$ of people. Although it works well for pose tracking, this approach is not a general solution for global localization. The reason is that sensor readings from people are correctly interpreted only with a correct hypothesis on the robot pose, but during global localization, there may be incorrect hypotheses that lead to incorrect interpretation of sensor readings. In other words, during global localization, there exists a dependence between the robot pose and positions of people, which is not captured by $Bel(s, \vec{u})$.

Hence we maintain a probability distribution $Bel(s, P_s(\vec{u}))$ of robot pose s and *probability distribution* $P_s(\vec{u})$ *of positions* $\vec{u}$ *of people conditioned on robot pose* s. $P_s(\vec{u})$ captures the dependence between the robot pose and positions of people. $Bel(s, P_s(\vec{u}))$ can be thought of as a probability distribution over probability distributions.

As in robot localization, we update $Bel(s, P_s(\vec{u}))$ with a Bayes filter. The difference from robot localization is that the state is a pair of s and $P_s(\vec{u})$ and that the action model takes as input both an action a and a measurement m. We use update equations (5) and (6) in Figure 9 (which are obtained by replacing s by $s, P_s(\vec{u})$ and a by a, m in update equations (1) and (2)).

The action model $\mathcal{A}(s, P_s(\vec{u})|a, m, s', P_{s'}(\vec{u'}))$ generates $s, P_s(\vec{u})$ from $s', P_{s'}(\vec{u'})$ utilizing action a and measurement m. We first generate s and then $P_s(\vec{u})$ according to equation (7) in Figure 9. We write the first *Prob* in equation (7) as $\mathcal{A}_{\mathsf{robot}}(s|a, m, s', P_{s'}(\vec{u'}))$. The second *Prob* in equation (7) indicates that we generate $P_s(\vec{u})$ from $P_{s'}(\vec{u'})$ utilizing action a and measurement m, which is exactly a situation where we can use another Bayes filter. For this inner Bayes filter, we use update equations (8) and (9) in Figure 9. We write *Prob* in equation (8) as $\mathcal{A}_{\mathsf{people}}(\vec{u}|a, \vec{u'}, s, s')$; we simplify *Prob* in equation (9) into $Prob(m|\vec{u}, s)$ because m

$$\begin{aligned}
Bel(s, P_s(\vec{u})) &\leftarrow \int \mathcal{A}(s, P_s(\vec{u})|a, m, s', P_{s'}(\vec{u'}))Bel(s', P_{s'}(\vec{u'}))d(s', P_{s'}(\vec{u'})) && (5)\\
Bel(s, P_s(\vec{u})) &\leftarrow \eta\mathcal{P}(m|s, P_s(\vec{u}))Bel(s, P_s(\vec{u})) && (6)\\
&= \eta Bel(s, P_s(\vec{u}))\int \mathcal{P}_{\mathsf{people}}(m|\vec{u}, s)P_s(\vec{u})d\vec{u} &&\\
\mathcal{A}(s, P_s(\vec{u})|a, m, s', P_{s'}(\vec{u'})) &= Prob(s|a, m, s', P_{s'}(\vec{u'}))\ Prob(P_s(\vec{u})|a, m, s', P_{s'}(\vec{u'}), s) && (7)\\
&= \mathcal{A}_{\mathsf{robot}}(s|a, m, s', P_{s'}(\vec{u'}))\ Prob(P_s(\vec{u})|a, m, s', P_{s'}(\vec{u'}), s) &&\\
P_s(\vec{u}) &\leftarrow \int Prob(\vec{u}|a, \vec{u'}, s, s')P_{s'}(\vec{u'})d\vec{u'} && (8)\\
&= \int \mathcal{A}_{\mathsf{people}}(\vec{u}|a, \vec{u'}, s, s')P_{s'}(\vec{u'})d\vec{u'} &&\\
P_s(\vec{u}) &\leftarrow \eta' Prob(m|\vec{u}, s, s')P_s(\vec{u}) && (9)\\
&= \eta'\mathcal{P}_{\mathsf{people}}(m|\vec{u}, s)P_s(\vec{u}) &&
\end{aligned}$$

Fig. 9. Equations used in people tracking. (5) and (6) for the Bayes filter computing $Bel(s, P_s(\vec{u}))$. (7) for decomposing the action model. (8) and (9) for the inner Bayes filter computing $P_s(\vec{u})$.

```
let $Bel_{new}$ =                                                              } (5)
  prob  sample $(s', P_{s'}(\vec{u'}))$ from $Bel$ in                           } (5)
        sample $s$ from $M_{\mathcal{A}_{\mathsf{robot}}}\ (a, m, s', P_{s'}(\vec{u'}))$ in     } (7)
        let $P_s(\vec{u})$ = prob  sample $\vec{u'}$ from $P_{s'}(\vec{u'})$ in              } (8)
                                   sample $\vec{u}$ from $M_{\mathcal{A}_{\mathsf{people}}}\ (a, \vec{u'}, s, s')$ in   } (8)
                                   $\vec{u}$                                                  } (8)
          in
        let $P_s(\vec{u})$ = bayes $(M_{\mathcal{P}_{\mathsf{people}}}\ m\ s)\ P_s(\vec{u})$ in      } (9)
        $(s, P_s(\vec{u}))$
let $Bel_{new}$ =                                                              } (6)
  bayes $\lambda(s, P_s(\vec{u}))$:_. (expectation $(M_{\mathcal{P}_{\mathsf{people}}}\ m\ s)\ P_s(\vec{u})$) $Bel$   } (6)
```

Fig. 10. Implementation of people tracking in $\lambda_{\bigcirc}$. Numbers on the right-hand side show corresponding equations in Figure 9.

does not depend on s' if s is given, and write it as $\mathcal{P}_{\mathsf{people}}(m|\vec{u}, s)$.

Figure 10 shows the implementation of people tracking in $\lambda_{\bigcirc}$. $M_{\mathcal{A}_{\mathsf{robot}}}$ and $M_{\mathcal{A}_{\mathsf{people}}}$ denote conditional probabilities $\mathcal{A}_{\mathsf{robot}}$ and $\mathcal{A}_{\mathsf{people}}$, respectively. $M_{\mathcal{P}_{\mathsf{people}}}\ m\ s$ returns a function $f(\vec{u}) = \kappa\mathcal{P}_{\mathsf{people}}(m|\vec{u}, s)$ for a constant κ. Since both m and s are fixed when computing $f(\vec{u})$, we consider only those range readings in m that correspond with people. In implementing update equation (6), we use the fact that $\mathcal{P}(m|s, P_s(\vec{u}))$ is the expectation of a function $g(\vec{u}) = \mathcal{P}_{\mathsf{people}}(m|\vec{u}, s)$ with respect to $P_s(\vec{u})$:

$$\mathcal{P}(m|s, P_s(\vec{u})) = \int \mathcal{P}_{\mathsf{people}}(m|\vec{u}, s)P_s(\vec{u})d\vec{u} \qquad (10)$$

Our implementation further simplifies the models used in the update equations. We use $\mathcal{A}_{\mathsf{robot}}(s|a, s')$ instead of $\mathcal{A}_{\mathsf{robot}}(s|a, m, s', P_{s'}(\vec{u'}))$ as in robot localization. That is, we ignore the interaction between the robot and people when generating new poses of the robot. Similarly we use $\mathcal{A}_{\mathsf{people}}(\vec{u}|\vec{u'})$ instead of $\mathcal{A}_{\mathsf{people}}(\vec{u}|a, \vec{u'}, s, s')$ on the assumption that positions of people are not affected by the robot pose; $\vec{u}$ is obtained by adding a random noise to $\vec{u'}$. We also simplify $\mathcal{P}(m|s, P_s(\vec{u}))$ in update equation (6) into $\mathcal{P}(m|s)$, which is computed in the same way as in robot localization; hence equation (10) is not actually exploited in our implementation.

```
let $Bel_{new}$ =                                                                          } (5)
  prob sample $(s', P_{s'}(\vec{u'}))$ from $Bel$ in                                          } (5)
    sample $s$ from                                                               } (7)   } (5)
      bayes $\lambda s$:_. (expectation $(M_{\mathcal{P}_{\text{landmark}}}\ m\ s)\ P_{s'}(\vec{u'})$)   } (11)  } (7)   } (5)
            $(M_{\mathcal{A}_{\text{robot}}}\ (a, s'))$ in                             } (11)  } (7)   } (5)
    let $P_s(\vec{u})$ = bayes $(M_{\mathcal{P}_{\text{landmark}}}\ m\ s)\ P_{s'}(\vec{u'})$ in                 } (9)   } (7)   } (5)
    $(s, P_s(\vec{u}))$                                                                 } (7)   } (5)
let $Bel_{new}$ = bayes $\lambda(s, P_s(\vec{u}))$:_. (expectation $(M_{\mathcal{P}_{\text{landmark}}}\ m\ s)\ P_s(\vec{u})$) $Bel$   } (6)
```

Fig. 11. Implementation of robotic mapping in $\lambda_{\bigcirc}$.

8.4 Robotic mapping

Robotic mapping [Thrun 2002] is the problem of building a map (or a spatial model) of the environment from sensor readings. Since measurements are a sequence of inaccurate local snapshots of the environment, a robot simultaneously localizes itself as it explores the environment so that it corrects and aligns local snapshots to construct a global map. For this reason, robotic mapping is also referred to as *simultaneous localization and mapping* (or SLAM).

Here we assume that the environment consists of an unknown number of stationary landmarks. Then the goal is to estimate positions of landmarks as well as the robot pose. The key observation is that we may think of landmarks as people who never move in an empty environment. It means that the problem is a special case of people tracking and we can use all the equations in Figure 9. Below we use subscript $_{\text{landmark}}$ instead of $_{\text{people}}$ for the sake of clarity.

As in people tracking, we maintain a probability distribution $Bel(s, P_s(\vec{u}))$ of robot pose s and probability distribution $P_s(\vec{u})$ of positions $\vec{u}$ of landmarks conditioned on robot pose s. Since landmarks are stationary and $\mathcal{A}_{\text{landmark}}(\vec{u}|a, \vec{u'}, s, s')$ is non-zero if and only if $\vec{u} = \vec{u'}$, we skip update equation (8) in implementing update equation (5). $\mathcal{A}_{\text{robot}}$ in equation (7) uses $\mathcal{P}_{\text{landmark}}(m|\vec{u'}, s)$ to test the likelihood of each new robot pose s with respect to old positions $\vec{u'}$ of landmarks, as in FastSLAM 2.0 [Montemerlo 2003]:

$$\begin{aligned}
& \mathcal{A}_{\text{robot}}(s|a, m, s', P_{s'}(\vec{u'})) & (11)\\
= \ & \int Prob(s|a, m, s', u')P_{s'}(\vec{u'})d\vec{u'} \\
= \ & \int \frac{Prob(s|a, \vec{u'})Prob(m, s'|s, a, \vec{u'})}{Prob(m, s'|a, \vec{u'})} P_{s'}(\vec{u'})d\vec{u'} \\
= \ & \int \eta'' Prob(m, s'|s, a, \vec{u'})P_{s'}(\vec{u'})d\vec{u'} \quad \textit{where} \quad \eta'' = \frac{Prob(s|a, \vec{u'})}{Prob(m, s'|a, \vec{u'})} \\
= \ & \int \eta'' Prob(s'|s, a, \vec{u'}, m)Prob(m|s, a, \vec{u'})P_{s'}(\vec{u'})d\vec{u'} \\
= \ & \int \eta'' Prob(s'|s, a)Prob(m|s, \vec{u'})P_{s'}(\vec{u'})d\vec{u'} \\
= \ & \eta'' \mathcal{A}_{\text{robot}}(s|a, s') \int \mathcal{P}_{\text{landmark}}(m|\vec{u'}, s)P_{s'}(\vec{u'})d\vec{u'}
\end{aligned}$$

Given a and s', we implement equation (11) with a Bayes operation on $\mathcal{A}_{\text{robot}}(\cdot|a, s')$.

Figure 11 shows the implementation of robotic mapping in $\lambda_{\bigcirc}$. Compared with the implementation of people tracking in Figure 10, it omits update equation (8) and incorporates equation (11). $M_{\mathcal{A}_{\mathsf{robot}}}$ and $M_{\mathcal{P}_{\mathsf{landmark}}}$ denote conditional probabilities $\mathcal{A}_{\mathsf{robot}}$ and $\mathcal{P}_{\mathsf{landmark}}$, respectively, as in people tracking. Since landmarks are stationary, we no longer need $M_{\mathcal{A}_{\mathsf{landmark}}}$. If we approximate $Bel(s, P_s(\vec{u}))$ with a single sample (*i.e.*, with one most likely robot pose and an associated map), update equation (6) becomes unnecessary.

8.5 Discussion

We find that the benefit of implementing probabilistic computations in $\lambda_{\bigcirc}$, such as improved readability and conciseness of code, can outweigh its disadvantage in speed. For example, our robot localizer is 1307 lines long (826 lines of Objective CAML/$\lambda_{\bigcirc}$ code for probabilistic computations and 481 lines of C code for interfacing with CARMEN [Montemerlo et al.]) whereas the CARMEN robot localizer, which uses particle filters and is written in C, is 3397 lines long. (Our robot localizer also uses the translator of $\lambda_{\bigcirc}$ which is 306 lines long: 53 lines of CAMLP4 code and 253 lines of Objective CAML code.) The comparison is, however, not conclusive because not every piece of code in CARMEN contributes to robot localization. Moreover the reduction in code size is also attributed to the use of Objective CAML as the host language. Hence the comparison should not be taken as indicative of reduction in code size due to $\lambda_{\bigcirc}$ alone. The speed loss is also not significant. For example, while the CARMEN robot localizer processes 100.0 sensor readings, our robot localizer processes on average 54.6 sensor readings (and nevertheless shows comparable accuracy).

On the other hand, $\lambda_{\bigcirc}$ is not suitable for an application that achieves high scalability by exploiting a particular representation scheme for probability distributions. In the robotic mapping problem, for example, one may choose to approximate the position of each landmark with a Gaussian distribution. As the cost of representing a Gaussian distribution is relatively low, the approximation makes it possible to build a highly scalable mapper. For example, Montemerlo [Montemerlo 2003] presents a FastSLAM 2.0 mapper which handles maps with over 1,000,000 landmarks. For such a problem, $\lambda_{\bigcirc}$ would be useful for quickly building a prototype implementation to test the correctness of a probabilistic computation.

9. RELATED WORK

There are a number of probabilistic languages that focus on discrete distributions. Such a language usually provides a probabilistic construct that is equivalent to a binary choice construct. Saheb-Djahromi [Saheb-Djahromi 1978] presents a probabilistic language with a binary choice construct $(p_1 \rightarrow e_1, p_2 \rightarrow e_2)$ where $p_1 + p_2 = 1.0$. Koller, McAllester, and Pfeffer [Koller et al. 1997] present a first order functional language with a coin toss construct $\mathsf{flip}(p)$. Pfeffer [Pfeffer 2001] generalizes the coin toss construct to a multiple choice construct $\mathsf{dist}\,[p_1 : e_1, \cdots, p_n : e_n]$ where $\sum_i p_i = 1.0$. Gupta, Jagadeesan, and Panangaden [Gupta et al. 1999] present a stochastic concurrent constraint language with a probabilistic choice construct $\mathsf{choose}\ x\ \mathsf{from}\ Dom\ \mathsf{in}\ e$ where Dom is a finite set of real numbers. All these constructs, although in different forms, are equivalent to a binary choice construct and have the same expressive power.

An easy way to compute a binary choice construct (or an equivalent) is to generate a sample from the probability distribution it denotes, as in the above probabilistic languages. Another way is to return an accurate representation of the probability distribution itself, by enumerating all elements in its support along with their probabilities. Pless and Luger [Pless and Luger 2001] present an extended lambda calculus which uses a probabilistic construct of the form $\sum_i e_i : p_i$ where $\sum_i p_i = 1.0$. An expression denoting a probability distribution computes to a normal form $\sum_i v_i : p_i$, which is an accurate representation of the probability distribution. Jones [Jones 1990] presents a metalanguage with a binary choice construct $e_1 \ \mathsf{or}_p \ e_2$. Its operational semantics uses a judgment $e \Rightarrow \sum p_i v_i$. Mogensen [Mogensen 2002] presents a language for specifying die-rolls. Its denotation semantics (called *probability semantics*) is formulated in a similar style, in terms of probability measures.

Jones and Mogensen also provide an equivalent of a fixed point construct which enables programmers to specify discrete distributions with infinite support (*e.g.*, geometric distribution). Jones assumes $\sum p_i \leq 1.0$ in the judgment $e \Rightarrow \sum p_i v_i$ and Mogensen uses *partial probability distributions* in which the sum of probabilities may be less than 1.0. The intuition is that we allow only a finite recursion depth so that some elements can be omitted in the enumeration.

There are a few probabilistic languages supporting continuous distributions. Kozen [Kozen 1981] investigates the semantics of probabilistic while programs. A random assignment $x := \mathsf{random}$ assigns a random number to variable x. Since it does not assume a specific probability distribution for the random number generator, the language serves only as a framework for probabilistic languages. The third author [Thrun 2000b] extends C++ with probabilistic data types which are created from a template prob<*type*>. Although the language supports common continuous distributions, its semantics is not formally defined. The first author [Park 2003] presents a probabilistic calculus whose mathematical basis is sampling functions. In order to encode sampling functions directly, the calculus uses a *sampling construct* $\gamma.e$ where γ is a formal argument and e denotes the body of a sampling function. As in $\lambda_{\bigcirc}$, the computation of $\gamma.e$ proceeds by generating a random number from $U(0.0, 1.0]$ and substituting it for γ in e.

The idea of using a monadic syntax in $\lambda_{\bigcirc}$ was inspired by Ramsey and Pfeffer [Ramsey and Pfeffer 2002]. They present a stochastic lambda calculus (with a binary choice construct $\mathsf{choose}\ p\ e_1\ e_2$) whose denotational semantics is based upon the monad of probability measures, or the probability monad [Giry 1981; Jones 1990]. In implementing a query for generating samples from probability distributions, they note that the probability monad can also be interpreted in terms of sampling functions, both denotationally and operationally. In designing $\lambda_{\bigcirc}$, we take the opposite approach: first we use a monadic syntax for probabilistic computations and relate it to sampling functions; then we interpret it in terms of probability distributions.

10. CONCLUSION

We have presented a probabilistic language $\lambda_{\bigcirc}$ whose mathematical basis is sampling functions. $\lambda_{\bigcirc}$ supports discrete distributions, continuous distributions, and even those belonging to neither group, without drawing a syntactic or semantic distinction. To the best of our knowledge, $\lambda_{\bigcirc}$ is the only probabilistic language with a formal semantics that has been applied to real problems involving continuous distributions. There are a few other probabilistic languages that are capable of simulating continuous distributions (by combining an infinite number of discrete distributions), but they require a special treatment such as the lazy evaluation strategy in [Koller et al. 1997; Pfeffer 2001] and the limiting process in [Gupta et al. 1999].

$\lambda_{\bigcirc}$ does not support precise reasoning about probability distributions in the sense that it is incapable of automatically inferring closed-form analytic solutions to queries on probability distributions encoded by arbitrary expressions. Note, however, that this is not an inherent limitation of $\lambda_{\bigcirc}$ due to its use of sampling functions as the mathematical basis; rather this is a necessary feature of $\lambda_{\bigcirc}$ because precise reasoning about probability distributions in our sense is impossible in general. In other words, if $\lambda_{\bigcirc}$ supported precise reasoning, it would support a much smaller set of probability distributions and operations.

The utility of a probabilistic language depends on each problem to which it is applied. $\lambda_{\bigcirc}$ is a good choice for those problems in which non-discrete distributions are used or precise reasoning is unnecessary. Robotics is a good example, since non-discrete distributions are used (even those probability distributions similar to *point_uniform* in Section 4 are used in modeling laser range finders) and also precise reasoning is unnecessary (sensor readings are inaccurate at any rate). On the other hand, $\lambda_{\bigcirc}$ may not be the best choice for those problems involving only discrete distributions, since its rich expressiveness is not fully exploited and approximate reasoning may be too weak for discrete distributions.

ACKNOWLEDGMENTS

We are grateful to the four anonymous referees for their insightful comments which helped us improve the paper significantly.

A. APPENDIX

A.1 Proof of Proposition 3.6

LEMMA A.1.
If $\Gamma \vdash F \div A$ *and* $\Gamma, \mathbf{x} \div A \vdash M : B$*, then* $\Gamma \vdash [F/\mathbf{x}]M : B$*.*
If $\Gamma \vdash F \div A$ *and* $\Gamma, \mathbf{x} \div A \vdash E \div B$*, then* $\Gamma \vdash [F/\mathbf{x}]E \div B$*.*

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

PROOF OF PROPOSITION 3.6. By simultaneous induction on the structure of the derivation of $\Gamma \vdash M : A$ and $\Gamma \vdash E \div A$. An interesting case is when $E =$ efix $\mathbf{x} \div A.\, F$.

Case $E =$ efix $\mathbf{x} \div A.\, F$:

$\Gamma, \mathbf{x} \div A \vdash F \div A$ by Efix

$\quad\Gamma, \mathbf{x} \div A \vdash F^\star \div A$ by induction hypothesis

$\quad\Gamma, x_p : \bigcirc A, \mathbf{x} \div A \vdash F^\star \div A$ by weakening

$\Gamma, x_p : \bigcirc A \vdash$ sample y_v from x_p in $y_v \div A$ (typing derivation)

$\Gamma, x_p : \bigcirc A \vdash$ [sample y_v from x_p in $y_v/\mathbf{x}]F^\star \div A$ by Lemma A.1

$\quad\Gamma \vdash$ sample y_r from fix $x_p{:}\bigcirc A.$ prob [sample y_v from x_p in $y_v/\mathbf{x}]F^\star$ in $y_r \div A$ (typing derivation)

$\Gamma \vdash (\mathsf{efix}\ \mathbf{x}\div A.\, F)^\star \div A$ by the definition of $(\cdot)^\star$ □

A.2 Proof of Propositions 3.7 and 3.8

PROPOSITION A.2.
For any term N, we have $([N/x]M)^\star = [N^\star/x]M^\star$ and $([N/x]E)^\star = [N^\star/x]E^\star$.
For any expression F, we have $([F/\mathbf{x}]M)^\star = [F^\star/\mathbf{x}]M^\star$ and $([F/\mathbf{x}]E)^\star = [F^\star/\mathbf{x}]E^\star$.

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

LEMMA A.3. *If $M \mapsto_{\mathsf{t}} N$, then $M^\star \mapsto_{\mathsf{t}} N^\star$.*

PROOF. By induction on the structure of the derivation of $M \mapsto_{\mathsf{t}} N$. □

LEMMA A.4.
If $M^\star \mapsto_{\mathsf{t}} N'$, then there exists N such that $N' = N^\star$ and $M \mapsto_{\mathsf{t}} N$.

PROOF. By induction on the structure of the derivation of $M^\star \mapsto_{\mathsf{t}} N'$. □

We introduce an equivalence relation $\equiv_{\mathsf{e}}$ on expressions to state that two expressions compute to the same value.

Definition A.5.
$E \equiv_{\mathsf{e}} F$ if and only if $E \mathbin{@} \omega \mapsto_{\mathsf{e}}^* V \mathbin{@} \omega'$ implies $F \mathbin{@} \omega \mapsto_{\mathsf{e}}^* V \mathbin{@} \omega'$, and vice versa.

The following equivalences are used in proofs below:

$$\begin{array}{rcl} \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E\ \mathsf{in}\ x & \equiv_{\mathsf{e}} & E \\ \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E\ \mathsf{in}\ F & \equiv_{\mathsf{e}} & \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E'\ \mathsf{in}\ F\ \ \textit{where}\ \ E \equiv_{\mathsf{e}} E' \\ (\mathsf{efix}\ \mathbf{x}\div A.\, E)^\star & \equiv_{\mathsf{e}} & [(\mathsf{efix}\ \mathbf{x}\div A.\, E)^\star/\mathbf{x}]E^\star \end{array}$$

The third equivalence follows from an expression reduction

$$(\mathsf{efix}\ \mathbf{x}\div A.\, E)^\star \mathbin{@} \omega \mapsto_{\mathsf{e}} \mathsf{sample}\ y_r\ \mathsf{from}\ \mathsf{prob}\ [(\mathsf{efix}\ \mathbf{x}\div A.\, E)^\star/\mathbf{x}]E^\star\ \mathsf{in}\ y_r \mathbin{@} \omega.$$

PROOF OF PROPOSITION 3.7. By induction on the structure of the derivation of $E \mathbin{@} \omega \mapsto_{\mathsf{e}} F \mathbin{@} \omega'$. We consider the case $E = \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E_0$ where $M \neq \mathsf{prob}\ E'$.

If $\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E_0 \mathbin{@} \omega \mapsto_{\mathsf{e}} \mathsf{sample}\ x\ \mathsf{from}\ N\ \mathsf{in}\ E_0 \mathbin{@} \omega$ by the rule E_{Bind}, then $M \mapsto_{\mathsf{t}} N$.

By Lemma A.3, $M^\star \mapsto_{\mathsf{t}} N^\star$.

Since

$$(\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E_0)^\star = \mathsf{sample}\ x\ \mathsf{from}\ M^\star\ \mathsf{in}\ {E_0}^\star$$

and

$$(\mathsf{sample}\ x\ \mathsf{from}\ N\ \mathsf{in}\ E_0)^\star = \mathsf{sample}\ x\ \mathsf{from}\ N^\star\ \mathsf{in}\ {E_0}^\star,$$

we have $(\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E_0)^\star \mathbin{@} \omega \mapsto_{\mathsf{e}} (\mathsf{sample}\ x\ \mathsf{from}\ N\ \mathsf{in}\ E_0)^\star \mathbin{@} \omega$.

Then we let $F' = (\mathsf{sample}\ x\ \mathsf{from}\ N\ \mathsf{in}\ E_0)^\star$. □

PROOF OF PROPOSITION 3.8. By induction on the structure of the derivation of $E^\star \mathbin{@} \omega \mapsto_{\mathsf{e}} F' \mathbin{@} \omega'$. An interesting case is when the rule E_{Bind} is applied last in a given derivation.

If $E = \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E_0$, then $E^\star = \mathsf{sample}\ x\ \mathsf{from}\ M^\star\ \mathsf{in}\ {E_0}^\star$.

By Lemma A.4, there exists N such that $M \mapsto_{\mathsf{t}} N$ and $M^\star \mapsto_{\mathsf{t}} N^\star$.
Hence we have

$$E \mathbin{@} \omega \mapsto_{\mathsf{e}} \mathsf{sample}\ x\ \mathsf{from}\ N\ \mathsf{in}\ E_0 \mathbin{@} \omega'$$

and

$$E^\star \mathbin{@} \omega \mapsto_{\mathsf{e}} \mathsf{sample}\ x\ \mathsf{from}\ N^\star\ \mathsf{in}\ {E_0}^\star \mathbin{@} \omega'$$

(where $\omega = \omega'$).
Then we let $F = \mathsf{sample}\ x\ \mathsf{from}\ N\ \mathsf{in}\ E_0$.
If $E = \mathsf{efix}\ \mathbf{x}{\div}A.\,E_0$, then $F' \equiv_{\mathsf{e}} ([\mathsf{efix}\ \mathbf{x}{\div}A.\,E_0/\mathbf{x}]E_0)^\star$ (and $\omega = \omega'$)
because $(\mathsf{efix}\ \mathbf{x}{\div}A.\,E_0)^\star \equiv_{\mathsf{e}} [(\mathsf{efix}\ \mathbf{x}{\div}A.\,E_0)^\star/\mathbf{x}]{E_0}^\star = ([\mathsf{efix}\ \mathbf{x}{\div}A.\,E_0/\mathbf{x}]E_0)^\star$.
Then we let $F = [\mathsf{efix}\ \mathbf{x}{\div}A.\,E_0/\mathbf{x}]E_0$. □

A.3 Proof of Proposition 7.2

Proof of Proposition 7.2. By simultaneous induction on the structure of M and E. The conclusion in the first clause also implies $\Gamma \vdash M \div B$. An interesting case is when $E = \mathbf{x}$.

Case $E = \mathbf{x}$:

$[\Gamma]_{\mathsf{v}} \vdash_{\mathsf{v}} [\mathbf{x}]_{\mathsf{v}} : A$ — by assumption
$[\Gamma]_{\mathsf{v}} \vdash_{\mathsf{v}} x_{\mathbf{x}}\ () : A$ — because $[\mathbf{x}]_{\mathsf{v}} = x_{\mathbf{x}}\ ()$
$x_{\mathbf{x}} : \mathtt{unit} \mathbin{\texttt{->}} A \in [\Gamma]_{\mathsf{v}}$ — by App and Unit
Since $x_{\mathbf{x}}$ is a special variable annotated with expression variable $\mathbf{x}$,
$x_{\mathbf{x}} \div B \in \Gamma$ and $A = [B]_{\mathsf{v}}$ for some B.
$A = [B]_{\mathsf{v}}$ and $\Gamma \vdash E \div B$. □

A.4 Proof of Propositions 7.3 and 7.4

Lemma A.6. *$[[M/x]N]_{\mathsf{v}} = [[M]_{\mathsf{v}}/x][N]_{\mathsf{v}}$ and $[[M/x]E]_{\mathsf{v}} = [[M]_{\mathsf{v}}/x][E]_{\mathsf{v}}$.*

Proof. By simultaneous induction on the structure of N and E. □

Lemma A.7.
$[[\mathsf{efix}\ \mathbf{x}{\div}A.\,E/\mathbf{x}]M]_{\mathsf{v}} = [(\mathtt{fix}\ x_{\mathbf{x}}{:}\mathtt{unit} \mathbin{\texttt{->}} [A]_{\mathsf{v}}.\ \mathtt{fun}\ \_{:}\mathtt{unit}.\ [E]_{\mathsf{v}})/x_{\mathbf{x}}][M]_{\mathsf{v}}$.
$[[\mathsf{efix}\ \mathbf{x}{\div}A.\,E/\mathbf{x}]F]_{\mathsf{v}} = [(\mathtt{fix}\ x_{\mathbf{x}}{:}\mathtt{unit} \mathbin{\texttt{->}} [A]_{\mathsf{v}}.\ \mathtt{fun}\ \_{:}\mathtt{unit}.\ [E]_{\mathsf{v}})/x_{\mathbf{x}}][F]_{\mathsf{v}}$.

Proof. By simultaneous induction on the structure of M and F. □

Corollary A.8.
$[[\mathsf{efix}\ \mathbf{x}{\div}A.\,E/\mathbf{x}]E]_{\mathsf{v}} = [(\mathtt{fix}\ x_{\mathbf{x}}{:}\mathtt{unit} \mathbin{\texttt{->}} [A]_{\mathsf{v}}.\ \mathtt{fun}\ \_{:}\mathtt{unit}.\ [E]_{\mathsf{v}})/x_{\mathbf{x}}][E]_{\mathsf{v}}$.

Proof of Proposition 7.3. By induction on the structure of the derivation of $M \mapsto_{\mathsf{t}} N$.

Case $\dfrac{M \mapsto_{\mathsf{t}} M'}{M\ N \mapsto_{\mathsf{t}} M'\ N}\ T_{\beta_L}$:

$[M]_{\mathsf{v}} \mathbin{@} \omega \mapsto_{\mathsf{v}} [M']_{\mathsf{v}} \mathbin{@} \omega$ — by induction hypothesis
$[M\ N]_{\mathsf{v}} = [M]_{\mathsf{v}}\ [N]_{\mathsf{v}}$
$[M]_{\mathsf{v}}\ [N]_{\mathsf{v}} \mathbin{@} \omega \mapsto_{\mathsf{v}} [M']_{\mathsf{v}}\ [N]_{\mathsf{v}} \mathbin{@} \omega$ — by E_{β_L}
$[M']_{\mathsf{v}}\ [N]_{\mathsf{v}} = [M'\ N]_{\mathsf{v}}$

Case $\dfrac{N \mapsto_{\mathsf{t}} N'}{(\lambda x{:}A.\,M)\ N \mapsto_{\mathsf{t}} (\lambda x{:}A.\,M)\ N'}\ T_{\beta_R}$:

$[N]_{\mathsf{v}} \mathbin{@} \omega \mapsto_{\mathsf{v}} [N']_{\mathsf{v}} \mathbin{@} \omega$ — by induction hypothesis

$[(\lambda x{:}A.\,M)\ N]_{\mathsf{v}} = (\texttt{fun}\ x{:}[A]_{\mathsf{v}}\texttt{.}\ [M]_{\mathsf{v}})\ [N]_{\mathsf{v}}$
$(\texttt{fun}\ x{:}[A]_{\mathsf{v}}\texttt{.}\ [M]_{\mathsf{v}})\ [N]_{\mathsf{v}}\ @\ \omega \mapsto_{\mathsf{v}} (\texttt{fun}\ x{:}[A]_{\mathsf{v}}\texttt{.}\ [M]_{\mathsf{v}})\ [N']_{\mathsf{v}}\ @\ \omega$ by $\mathtt{E}_{\beta_{\mathtt{R}}}$
$(\texttt{fun}\ x{:}[A]_{\mathsf{v}}\texttt{.}\ [M]_{\mathsf{v}})\ [N']_{\mathsf{v}} = [(\lambda x{:}A.\,M)\ N']_{\mathsf{v}}$

Case $\dfrac{}{(\lambda x{:}A.\,M)\ V \mapsto_{\mathsf{t}} [V/x]M}\ T_{\beta_V}$:
$[(\lambda x{:}A.\,M)\ V]_{\mathsf{v}} = (\texttt{fun}\ x{:}[A]_{\mathsf{v}}\texttt{.}\ [M]_{\mathsf{v}})\ [V]_{\mathsf{v}}$
$(\texttt{fun}\ x{:}[A]_{\mathsf{v}}\texttt{.}\ [M]_{\mathsf{v}})\ [V]_{\mathsf{v}}\ @\ \omega \mapsto_{\mathsf{v}} [[V]_{\mathsf{v}}/x][M]_{\mathsf{v}}\ @\ \omega$ by $\mathtt{E}_{\beta_{\mathtt{V}}}$
$[[V]_{\mathsf{v}}/x][M]_{\mathsf{v}} = [[V/x]M]_{\mathsf{v}}$ by Lemma A.6 □

Proof of Proposition 7.4. By induction on the structure of the derivation of $E\ @\ \omega \mapsto_{\mathsf{e}} F\ @\ \omega'$. We consider two interesting cases.

Case $\dfrac{E\ @\ \omega \mapsto_{\mathsf{e}} E'\ @\ \omega'}{\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E\ \mathsf{in}\ F\ @\ \omega \mapsto_{\mathsf{e}} \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E'\ \mathsf{in}\ F\ @\ \omega'}\ E_{BindR}$:
$[E]_{\mathsf{v}}\ @\ \omega \mapsto^*_{\mathsf{v}} e\ @\ \omega'$ where $[E']_{\mathsf{v}}\ @\ \omega' \mapsto^*_{\mathsf{v}} e\ @\ \omega'$ by induction hypothesis
$[\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E\ \mathsf{in}\ F]_{\mathsf{v}} = (\texttt{fun}\ x{:}\_\texttt{.}\ [F]_{\mathsf{v}})\ (\texttt{app}\ (\texttt{prb}\ (\texttt{fun}\ \_{:}\texttt{unit.}\ [E]_{\mathsf{v}})))$
$(\texttt{fun}\ x{:}\_\texttt{.}\ [F]_{\mathsf{v}})\ (\texttt{app}\ (\texttt{prb}\ (\texttt{fun}\ \_{:}\texttt{unit.}\ [E]_{\mathsf{v}})))\ @\ \omega$
$\mapsto_{\mathsf{v}} (\texttt{fun}\ x{:}\_\texttt{.}\ [F]_{\mathsf{v}})\ ((\texttt{fun}\ \_{:}\texttt{unit.}\ [E]_{\mathsf{v}})\ ())\ @\ \omega$ by $\mathtt{E}_{\mathtt{AppPrb}}$
$\mapsto_{\mathsf{v}} (\texttt{fun}\ x{:}\_\texttt{.}\ [F]_{\mathsf{v}})\ [E]_{\mathsf{v}}\ @\ \omega$ by $\mathtt{E}_{\beta_{\mathtt{V}}}$
$\mapsto^*_{\mathsf{v}} (\texttt{fun}\ x{:}\_\texttt{.}\ [F]_{\mathsf{v}})\ e\ @\ \omega'$ by $[E]_{\mathsf{v}}\ @\ \omega \mapsto^*_{\mathsf{v}} e\ @\ \omega'$
$[\mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ E'\ \mathsf{in}\ F]_{\mathsf{v}} = (\texttt{fun}\ x{:}\_\texttt{.}\ [F]_{\mathsf{v}})\ (\texttt{app}\ (\texttt{prb}\ (\texttt{fun}\ \_{:}\texttt{unit.}\ [E']_{\mathsf{v}})))$
$(\texttt{fun}\ x{:}\_\texttt{.}\ [F]_{\mathsf{v}})\ (\texttt{app}\ (\texttt{prb}\ (\texttt{fun}\ \_{:}\texttt{unit.}\ [E']_{\mathsf{v}})))\ @\ \omega'$
$\mapsto^*_{\mathsf{v}} (\texttt{fun}\ x{:}\_\texttt{.}\ [F]_{\mathsf{v}})\ [E']_{\mathsf{v}}\ @\ \omega'$ by $\mathtt{E}_{\mathtt{AppPrb}}$ and $\mathtt{E}_{\beta_{\mathtt{V}}}$
$\mapsto^*_{\mathsf{v}} (\texttt{fun}\ x{:}\_\texttt{.}\ [F]_{\mathsf{v}})\ e\ @\ \omega'$ by $[E']_{\mathsf{v}}\ @\ \omega' \mapsto^*_{\mathsf{v}} e\ @\ \omega'$

Case $\dfrac{}{\mathsf{efix}\ \mathbf{x}{\div}A.\,E\ @\ \omega \mapsto_{\mathsf{e}} [\mathsf{efix}\ \mathbf{x}{\div}A.\,E/\mathbf{x}]E\ @\ \omega}\ \mathit{Efix}$:
$[\mathsf{efix}\ \mathbf{x}{\div}A.\,E]_{\mathsf{v}} = (\texttt{fix}\ x_{\mathbf{x}}{:}\texttt{unit -> }[A]_{\mathsf{v}}\texttt{. fun}\ \_{:}\texttt{unit.}\ [E]_{\mathsf{v}})\ ()$
$(\texttt{fix}\ x_{\mathbf{x}}{:}\texttt{unit -> }[A]_{\mathsf{v}}\texttt{. fun}\ \_{:}\texttt{unit.}\ [E]_{\mathsf{v}})\ ()\ @\ \omega$
$\mapsto_{\mathsf{v}} (\texttt{fun}\ \_{:}\texttt{unit.}\ [\texttt{fix}\ x_{\mathbf{x}}{:}\texttt{unit -> }[A]_{\mathsf{v}}\texttt{. fun}\ \_{:}\texttt{unit.}\ [E]_{\mathsf{v}}/x_{\mathbf{x}}][E]_{\mathsf{v}})\ ()\ @\ \omega$
by $\mathtt{E}_{\mathtt{Fix}}$
$\mapsto^*_{\mathsf{v}} [\texttt{fix}\ x_{\mathbf{x}}{:}\texttt{unit -> }[A]_{\mathsf{v}}\texttt{. fun}\ \_{:}\texttt{unit.}\ [E]_{\mathsf{v}}/x_{\mathbf{x}}][E]_{\mathsf{v}}\ @\ \omega$ by $\mathtt{E}_{\beta_{\mathtt{V}}}$
$[[\mathsf{efix}\ \mathbf{x}{\div}A.\,E/\mathbf{x}]E]_{\mathsf{v}} = [\texttt{fix}\ x_{\mathbf{x}}{:}\texttt{unit -> }[A]_{\mathsf{v}}\texttt{. fun}\ \_{:}\texttt{unit.}\ [E]_{\mathsf{v}}/x_{\mathbf{x}}][E]_{\mathsf{v}}$
by Corollary A.8 □

A.5 Proof of Propositions 7.5 and 7.7

Proof of Proposition 7.5. By induction on the structure of M. We only need to consider the case $M = M_1\ M_2$. There are three cases of the structure of $[M_1\ M_2]_{\mathsf{v}}\ @\ \omega \mapsto_{\mathsf{v}} e\ @\ \omega'$ (corresponding to the rules $\mathtt{E}_{\beta_{\mathtt{L}}}$, $\mathtt{E}_{\beta_{\mathtt{R}}}$, and $\mathtt{E}_{\beta_{\mathtt{V}}}$). The case for the rule $\mathtt{E}_{\beta_{\mathtt{V}}}$ uses Lemma A.6. □

Proof of Proposition 7.7. By induction on the structure of E. We consider two cases $E = M$ and $E = \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ F$; the remaining cases are all trivial.

Case $E = M$, $[E]_{\mathsf{v}} = [M]_{\mathsf{v}}$:
$M \mapsto_{\mathsf{t}} \bullet$ by E_{Term}
$[M]_{\mathsf{v}}\ @\ \omega \mapsto_{\mathsf{v}} \bullet$ by Corollary 7.6
We let $e = [M]_{\mathsf{v}}$.

Case $E = \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ F$, $[E]_{\mathsf{v}} = (\texttt{fun}\ x{:}\_\texttt{.}\ [F]_{\mathsf{v}})\ \mathsf{app}\ [M]_{\mathsf{v}}$:
If $M \neq \mathsf{prob}\ \cdot$,
$M \mapsto_{\mathsf{t}} \bullet$ by E_{Bind}

$[M]_{\mathtt{v}}$ @ $\omega \mapsto_{\mathtt{v}} \bullet$ by Corollary 7.6

The rule $\mathtt{E_{App}}$ does not apply to $[E]_{\mathtt{v}}$.

The rule $\mathtt{E_{AppPrb}}$ does not apply to $[E]_{\mathtt{v}}$. $[M]_{\mathtt{v}} \neq \mathtt{prb} \cdot$

We let $e = [E]_{\mathtt{v}}$.

If $M = \mathsf{prob}\ E'$, $E' \neq V$,

E' @ $\omega \mapsto_{\mathtt{e}} \bullet$ by E_{BindR}

There exists e' such that $[E']_{\mathtt{v}}$ @ $\omega \mapsto_{\mathtt{v}}^{*} e'$ @ $\omega \mapsto_{\mathtt{v}} \bullet$ by induction hypothesis.

$[E]_{\mathtt{v}}$ @ ω

$\mapsto_{\mathtt{v}}^{*} (\mathtt{fun}\ x{:}\_.\ [F]_{\mathtt{v}})\ [E']_{\mathtt{v}}$ @ ω $[M]_{\mathtt{v}} = \mathtt{prb\ fun\ \_{:}unit.}\ [E']_{\mathtt{v}}$

$\mapsto_{\mathtt{v}}^{*} (\mathtt{fun}\ x{:}\_.\ [F]_{\mathtt{v}})\ e'$ @ ω $[E']_{\mathtt{v}}$ @ $\omega \mapsto_{\mathtt{v}}^{*} e'$ @ ω

$\mapsto_{\mathtt{v}} \bullet$ e' @ $\omega \mapsto_{\mathtt{v}} \bullet$

We let $e = (\mathtt{fun}\ x{:}\_.\ [F]_{\mathtt{v}})\ e'$.

If $M = \mathsf{prob}\ V$, then E @ $\omega \mapsto_{\mathtt{e}} \bullet$ does not hold because of the rule E_{BindV}. $\square$

REFERENCES

Bratley, P., Fox, B., and Schrage, L. 1996. *A guide to simulation*, 2nd ed. Springer-Verlag.

Charniak, E. 1993. *Statistical Language Learning.* MIT Press, Cambridge, Massachusetts.

Devroye, L. 1986. *Non-Uniform Random Variate Generation.* Springer-Verlag.

Doucet, A., de Freitas, N., and Gordon, N. 2001. *Sequential Monte Carlo Methods in Practice.* Springer-Verlag, New York.

Fox, D., Burgard, W., and Thrun, S. 1999. Markov localization for mobile robots in dynamic environments. *Journal of Artificial Intelligence Research 11*, 391–427.

Gill, J. 1977. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing 6,* 4, 675–695.

Giry, M. 1981. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, B. Banaschewski, Ed. Lecture Notes In Mathematics, vol. 915. Springer-Verlag, 68–85.

Gupta, V., Jagadeesan, R., and Panangaden, P. 1999. Stochastic processes as concurrent constraint programs. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, 189–202.

Henrion, M. 1988. Propagation of uncertainty in Bayesian networks by probabilistic logic sampling. In *Uncertainty in Artificial Intelligence 2*, J. F. Lemmer and L. N. Kanal, Eds. Elsevier/North-Holland, 149–163.

Isard, M. and Blake, A. 1998. CONDENSATION: conditional density propagation for visual tracking. *International Journal of Computer Vision 29,* 1, 5–28.

Jazwinski, A. H. 1970. *Stochastic Processes and Filtering Theory.* Academic Press, New York.

Jelinek, F. 1998. *Statistical Methods for Speech Recognition (Language, Speech, and Communication).* MIT Press, Boston, MA.

Jones, C. 1990. Probabilistic non-determinism. Ph.D. thesis, Department of Computer Science, University of Edinburgh.

Koller, D., McAllester, D., and Pfeffer, A. 1997. Effective Bayesian inference for stochastic programs. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97).* AAAI Press, 740–747.

Kozen, D. 1981. Semantics of probabilistic programs. *Journal of Computer and System Sciences 22,* 3, 328–350.

Launchbury, J. and Peyton Jones, S. L. 1995. State in Haskell. *Lisp and Symbolic Computation 8,* 4 (Dec.), 293–341.

Leroy, X. 2000. A modular module system. *Journal of Functional Programming 10,* 3, 269–303.

MacKay, D. J. C. 1998. Introduction to Monte Carlo methods. In *Learning in Graphical Models*, M. I. Jordan, Ed. NATO Science Series. Kluwer Academic Press, 175–204.

Martin-Löf, P. 1996. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic 1,* 1, 11–60. Text of lectures originally given in 1983 and distributed in 1985.

Mogensen, T. 2002. Roll: A language for specifying die-rolls. In *5th International Symposium on Practical Aspects of Declarative Languages*, V. Dahl and P. Wadler, Eds. LNCS, vol. 2562. Springer, 145–159.

Moggi, E. 1989. Computational lambda-calculus and monads. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 14–23.

Moggi, E. 1991. Notions of computation and monads. *Information and Computation 93*, 55–92.

Montemerlo, M. 2003. FastSLAM: A factored solution to the simultaneous localization and mapping problem with unknown data association. Ph.D. thesis, Robotics Institute, Carnegie Mellon University.

Montemerlo, M., Roy, N., and Thrun, S. CARMEN: Carnegie Mellon Robot Navigation Toolkit. `http://www.cs.cmu.edu/~carmen/`.

Montemerlo, M., Whittaker, W., and Thrun, S. 2002. Conditional particle filters for simultaneous mobile robot localization and people-tracking. In *IEEE International Conference on Robotics and Automation (ICRA)*. ICRA, Washington, DC, 695–701.

Motwani, R. and Raghavan, P. 1995. *Randomized Algorithms.* Cambridge Univ. Press. Motwani.

Park, S. 2003. A calculus for probabilistic languages. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation.* ACM Press, 38–49.

Peyton Jones, S. L. 2001. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering Theories of Software Construction*, C. A. R. Hoare, M. Broy, and R. Steinbrüggen, Eds. IOS Press, Amsterdam.

Peyton Jones, S. L. and Wadler, P. 1993. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, 71–84.

Pfeffer, A. 2001. IBAL: A probabilistic rational programming language. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, B. Nebel, Ed. Morgan Kaufmann Publishers, Inc., 733–740.

Pfenning, F. and Davies, R. 2001. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science 11,* 4, 511–540.

Pless, D. and Luger, G. 2001. Toward general analysis of recursive probability models. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence (UAI-01)*, J. Breese and D. Koller, Eds. Morgan Kaufmann Publishers, 429–436.

Rabiner, L. R. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE 77,* 2 (Feb.), 257–285.

Ramsey, N. and Pfeffer, A. 2002. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM Press, 154–165.

Rudin, W. 1986. *Real and Complex Analysis*, 3 ed. McGraw-Hill, New York.

Russell, S. and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach.* Prentice Hall.

Sabry, A. and Wadler, P. 1997. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems 19,* 6, 916–941.

Saheb-Djahromi, N. 1978. Probabilistic LCF. In *Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science*, J. Winkowski, Ed. LNCS, vol. 64. Springer, 442–451.

Thrun, S. 2000a. Probabilistic algorithms in robotics. *AI Magazine 21,* 4, 93–109.

Thrun, S. 2000b. Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. IEEE.

Thrun, S. 2002. Robotic mapping: A survey. In *Exploring Artificial Intelligence in the New Millenium*, G. Lakemeyer and B. Nebel, Eds. Morgan Kaufmann.

Welch, G. and Bishop, G. 1995. An introduction to the Kalman filter. Tech. Rep. TR95-041, Department of Computer Science, University of North Carolina - Chapel Hill.