A Probabilistic Language based upon Sampling Functions

Sungwoo Park **Frank Pfenning** **Sebastian Thrun**

November 1, 2004
CMU-CS-04-173

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

As probabilistic computations play an increasing role in solving various problems, researchers have designed probabilistic languages that treat probability distributions as primitive datatypes. Most probabilistic languages, however, focus only on discrete distributions and have limited expressive power. In this paper, we present a probabilistic language, called $\lambda_{\bigcirc}$, which uniformly supports all kinds of probability distributions – discrete distributions, continuous distributions, and even those belonging to neither group. Its mathematical basis is sampling functions, *i.e.*, mappings from the unit interval $(0.0, 1.0]$ to probability domains. We also briefly describe the implementation of $\lambda_{\bigcirc}$ as an extension of Objective CAML and demonstrate its practicality with three applications in robotics: robot localization, people tracking, and robotic mapping. All experiments have been carried out with real robots.

Sebastian Thrun is currently at Computer Science Department, Stanford University.

Keywords: Probabilistic language, Probability distribution, Sampling function, Robotics

1 Introduction

As probabilistic computations play an increasing role in solving various problems, researchers have designed *probabilistic languages* to facilitate their modeling [11, 7, 29, 22, 25, 15, 21]. A probabilistic language treats probability distributions as primitive datatypes and abstracts from their representation schemes. As a result, it enables programmers to concentrate on how to formulate probabilistic computations at the level of probability distributions rather than representation schemes. The translation of such a formulation in a probabilistic language usually produces concise and elegant code.

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 in the probability domain and its probability. We can use such a probabilistic language for problems involving only discrete distributions. For those involving non-discrete distributions, 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 weighted samples). For this reason, there has been little effort to develop probabilistic languages whose expressive power is beyond discrete distributions.

Our work aims to develop a probabilistic language supporting all kinds of probability distributions – discrete distributions, continuous distributions, and even those belonging to neither group. Furthermore we want to draw no distinction between different kinds of probability distributions, both syntactically and semantically, so that we can achieve a uniform framework for probabilistic computation. Such a probabilistic language can have a significant practical impact, since once formulated at the level of probability distributions, any probabilistic computation can be directly translated into code.

The main idea in our work is that we can specify any probability distribution 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 a uniform distribution over $(0.0, 1.0]$, it returns a sample in $\mathcal{D}$, and thus specifies a unique probability distribution. For our purpose, 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 it takes as input an infinite sequence of random numbers drawn independently from a uniform distribution over $(0.0, 1.0]$, consumes zero or more random numbers, and returns a sample with the remaining sequence.

We present a probabilistic language, called $\lambda_\bigcirc$, whose mathematical basis is sampling functions. We exploit the fact that sampling functions form a state monad, and base the syntax of $\lambda_\bigcirc$ upon the monadic metalanguage [17] in the formulation of Pfenning and Davies [23]. A syntactic distinction is drawn between regular values and probabilistic computations through the use of two syntactic categories: *terms* for regular values and *expressions* for probabilistic computations. It enables us to treat probability distributions as first-class values, and $\lambda_\bigcirc$ arises as a conservative extension of a conventional language. Examples show that $\lambda_\bigcirc$ provides a unified representation scheme for probability distributions, enjoys rich expressiveness, and allows high versatility in encoding probability distributions.

An important aspect of our work is to demonstrate the practicality of $\lambda_\bigcirc$ by applying it to real problems. As the main testbed, we choose *robotics* [28]. It offers a variety of real problems that necessitate probabilistic computations over continuous distributions. We implement $\lambda_\bigcirc$ as an extension of Objective CAML and use it for three applications in robotics: robot localization [28], people tracking [20], and robotic mapping [30]. We use real robots for all experiments.

$\lambda_\bigcirc$ does not support precise reasoning about probability distributions in that it does not permit a precise implementation of queries on probability distributions (such as expectation). This is in fact a feature of probability distributions that precise reasoning is impossible in general. In other words, lack of support for precise reasoning is the price we pay for rich expressiveness of $\lambda_\bigcirc$. As a practical solution, we use the

Monte Carlo method to support approximate reasoning. As such, $\lambda_{\bigcirc}$ is a good choice for those problems in which all kinds of probability distributions are used or precise reasoning is unnecessary or impossible.

This paper is organized as follows. Section 2 gives a motivating example for $\lambda_{\bigcirc}$. 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 Section 5 shows how to formally prove the correctness of encodings, based upon the operational semantics. Section 6 demonstrates the use of the Monte Carlo method for approximate reasoning and briefly describes our implementation of $\lambda_{\bigcirc}$. Section 7 presents three applications of $\lambda_{\bigcirc}$ in robotics. Section 8 discusses related work and Section 9 concludes. Appendix shows figures from experiments in Section 7.

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 $P(x)$"). If we do not need a specific name for a probability distribution, we use $Prob$. 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. $U(0.0, 1.0]$ denotes a uniform distribution over the unit interval $(0.0, 1.0]$.

2 A Motivating Example

A *Bayes filter* [9] is a popular solution to a wide range of state estimation problems. It estimates the state s of a system from a sequence of actions and measurements, where an action a makes a change to the state and a measurement m gives information on the state. At its core, a Bayes filter computes a probability distribution $Bel(s)$ of the 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 $\int Bel(s)ds = 1.0$. The update equations are formulated at the level of probability distributions in the sense that they do not assume a particular representation scheme.

Unfortunately the update equations are difficult to implement for arbitrary probability distributions. 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 instance, with the assumption that Bel is a Gaussian distribution, we obtain a variant of the Bayes filter called a *Kalman filter* [31]. If we approximate Bel with a set of samples, we obtain another variant called a *particle filter* [3].

Even these variants of the Bayes filter are, however, not trivial to implement in conventional languages, not to mention the difficulty of understanding the code. For instance, a Kalman filter requires various matrix operations including matrix inversion. A particle filter needs to manipulate weights associated with individual samples, which often results in complicated code.

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, we cannot easily choose an appropriate subset of its support upon which we perform discretization. Even when such a subset is fixed in advance, the process of discretization may require a considerable amount of programming; see [4] for an example. Second there are some probability distributions that cannot

be discretized in any meaningful way. An example is probability distributions over probability distributions, which do occur in real applications (see Section 7). Another example is probability distributions over function spaces.

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. We present such a probabilistic language $\lambda_{\bigcirc}$ in the next section.

3 Probabilistic Language $\lambda_{\bigcirc}$

In this section, we develop our probabilistic language $\lambda_{\bigcirc}$. We begin by explaining why we choose sampling functions as the mathematical basis of $\lambda_{\bigcirc}$.

3.1 Mathematical basis

The expressive power of a probabilistic language is determined to a large extent by its mathematical basis, *i.e.*, which mathematical objects are used to specify probability distributions. Since we intend to support all kinds of probability distributions without drawing a syntactic or semantic distinction, we cannot choose what is applicable only to a specific kind of probability distributions (*e.g.*, probability mass functions, probability density functions, or cumulative distribution functions). Probability measures are a possibility because they are synonymous with probability distributions. They are, however, not a practical choice: a probability measure on a domain $\mathcal{D}$ maps not $\mathcal{D}$ but the set of events on $\mathcal{D}$ to $[0.0, 1.0]$, and may be difficult to represent if $\mathcal{D}$ is an infinite domain.

Sampling functions overcome the problem with probability measures: they are applicable to all kinds of probability distributions, and are also easy to represent because a global random number generator supplants the use of infinite sequences of random numbers. For this reason, we choose sampling functions as the mathematical basis of $\lambda_{\bigcirc}$.[1]

It is noteworthy that sampling functions form a state monad [16, 17] whose set of states is $(0.0, 1.0]^{\infty}$. Moreover sampling functions are operationally equivalent to probabilistic computations because they describe procedures for generating samples. These two observations imply that if we use a monadic syntax for probabilistic computations, it becomes straightforward to interpret probabilistic computations in terms of sampling functions. Hence we use a monadic syntax for probabilistic computations in $\lambda_{\bigcirc}$.

3.2 Syntax and type system

As the linguistic framework of $\lambda_{\bigcirc}$, we use the monadic metalanguage of Pfenning and Davies [23]. It is a reformulation of Moggi's monadic metalanguage λ_{ml} [17], following Martin-Löf's methodology of distinguishing judgments from propositions [14]. 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. 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 as variables. $\lambda x : A.\, M$ is a lambda abstraction, and $M\ M$ is an application term. (M, M) is a product term, and $\mathsf{fst}\ M$ and $\mathsf{snd}\ M$ are projection terms; we include these terms in order to support joint distributions. $\mathsf{fix}\ x : A.\, M$ is a fixed point construct for recursive terms. A *probability term* $\mathsf{prob}\ E$ encapsulates an expression E; it is a first-class value denoting a

[1] Note, however, that not every sampling function specifies a probability distribution. For instance, no probability distribution is specified by a sampling function mapping rational numbers to 0 and irrational numbers to 1. Thus we restrict ourselves to those sampling functions that determine probability distributions (*i.e.*, *measurable* sampling functions).

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

Figure 1: Abstract syntax for $\lambda_{\bigcirc}$

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

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

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

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

Figure 2: Typing rules of $\lambda_{\bigcirc}$

probability distribution. Real numbers r are implemented as floating point numbers, since the overhead of exact real arithmetic is not justified in the domain where we work with samples and approximations anyway.

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

The type system employs a term typing judgment $\Gamma \vdash M : A$ and an expression typing judgment $\Gamma \vdash E \div A$ (Figure 2). $\Gamma \vdash M : A$ means that M evaluates to a value of type A under typing context Γ, and $\Gamma \vdash E \div A$ that E computes to a sample of type A under typing context Γ. The rule Prob is the introduction rule for the type constructor $\bigcirc$; it shows 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. All the remaining rules are standard.

3.3 Operational semantics

Since $\lambda_{\bigcirc}$ draws a syntactic distinction between regular values and probabilistic computations, its operational semantics needs two judgments: one for term evaluations and another for expression computations. A term evaluation is always deterministic and the corresponding judgment involves only terms. In contrast, an expression computation may consume random numbers and the corresponding judgment involves not only expressions but also sampling sequences. Since an expression computation may invoke term evaluations (*e.g.*, to evaluate M in $\mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E$), we first present the judgment for term evaluations and then use it for the judgment for expression computations. Both judgments use capture-avoiding substitutions $[N/x]M$ and $[N/x]E$ defined in a standard way.

For term evaluations, we introduce a judgment $M \mapsto N$ 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 use *evaluation contexts* which are terms with a hole $[]$ indicating where a term reduction may occur. We use $M \mapsto_{\mathsf{R}} N$ for term reductions:

$$\begin{array}{rrcl} \text{evaluation context} & \kappa & ::= & [] \mid \kappa\ M \mid (\lambda x{:}A.\, M)\ \kappa \mid \\ & & & (\kappa, M) \mid (V, \kappa) \mid \mathsf{fst}\ \kappa \mid \mathsf{snd}\ \kappa \end{array}$$

$$\begin{array}{rcl} (\lambda x{:}A.\, M)\ V & \mapsto_{\mathsf{R}} & [V/x]M \\ \mathsf{fst}\ (V_1, V_2) & \mapsto_{\mathsf{R}} & V_1 \\ \mathsf{snd}\ (V_1, V_2) & \mapsto_{\mathsf{R}} & V_2 \\ \mathsf{fix}\, x{:}A.\, M & \mapsto_{\mathsf{R}} & [\mathsf{fix}\, x{:}A.\, M/x]M \end{array} \qquad \frac{M \mapsto_{\mathsf{R}} N}{\kappa[M] \mapsto \kappa[N]}$$

We use $M \mapsto^* V$ for a term evaluation where $\mapsto^*$ denotes the reflexive and transitive closure of $\mapsto$. A term evaluation is always deterministic.

For expression computations, we introduce a judgment $E \mathbin{@} s \Rightarrow F \mathbin{@} s'$ which means that the computation of E with sampling sequence s reduces to the computation of F with sampling sequence s'. It uses *computation contexts* which are expressions with either a term hole $[]_{\mathsf{term}}$ or an expression hole $[]_{\mathsf{exp}}$. $[]_{\mathsf{term}}$ expects a term and $[]_{\mathsf{exp}}$ expects an expression. We use $E \mathbin{@} s \Rightarrow_{\mathsf{R}} F \mathbin{@} s'$ for expression reductions:

$$\begin{array}{rrcl} \text{computation context} & \iota & ::= & []_{\mathsf{exp}} \mid []_{\mathsf{term}} \mid \\ & & & \mathsf{sample}\ x\ \mathsf{from}\ []_{\mathsf{term}}\ \mathsf{in}\ E \mid \\ & & & \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ \iota\ \mathsf{in}\ E \end{array}$$

$$\begin{array}{rcl} \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ V\ \mathsf{in}\ E \mathbin{@} s & \Rightarrow_{\mathsf{R}} & [V/x]E \mathbin{@} s \\ \mathcal{S} \mathbin{@} rs & \Rightarrow_{\mathsf{R}} & r \mathbin{@} s \end{array}$$

$$\frac{M \mapsto N}{\iota[M]_{\mathsf{term}} \mathbin{@} s \Rightarrow \iota[N]_{\mathsf{term}} \mathbin{@} s} \qquad \frac{E \mathbin{@} s \Rightarrow_{\mathsf{R}} F \mathbin{@} s'}{\iota[E]_{\mathsf{exp}} \mathbin{@} s \Rightarrow \iota[F]_{\mathsf{exp}} \mathbin{@} s'}$$

We use $E \mathbin{@} s \Rightarrow^* V \mathbin{@} s'$ for an expression computation where $\Rightarrow^*$ denotes the reflexive and transitive closure of $\Rightarrow$. Note that an expression computation itself is deterministic; it is only when we vary sampling sequences that an expression exhibits probabilistic behavior.

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

Proposition 3.1. *If $E \mathbin{@} s \Rightarrow^* V \mathbin{@} s'$, then $s = r_1 r_2 \cdots r_n s'$ ($n \geq 0$) where*

$$E \mathbin{@} s \Rightarrow^* E_1 \mathbin{@} r_2 \cdots r_n s' \Rightarrow^* \cdots \Rightarrow^* E_{n-1} \mathbin{@} r_n s' \Rightarrow^* V \mathbin{@} s'$$

for a sequence of expressions $E_1, \cdots, E_{n-1}$.

Thus an expression computation coincides with the operational description of a sampling function when applied to a sampling sequence, which implies that an expression represents a sampling function.

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 $M \mapsto N$ *and* $\cdot \vdash M : A$*, then* $\cdot \vdash N : A$*.*
If $E \mathbin{@} s \Rightarrow F \mathbin{@} s'$ *and* $\cdot \vdash E \div A$*, then* $\cdot \vdash F \div A$*.*

Proof. For the first clause, we let $M = \kappa[M']$ and proceed by induction on the structure of κ. For the second clause, we let $E = \iota[M']_{\mathsf{term}}$ or $E = \iota[E']_{\mathsf{exp}}$ and proceed by induction on the structure of ι; we use the result from the first clause. □

Lemma 3.4 (Canonical forms). *If* $\cdot \vdash V : A$*, then:*
(1) if $A = A_1 \to A_2$*, 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* N *such that* $M \mapsto N$*.*
If $\cdot \vdash E \div A$*, then either* E *is a sample (*i.e.*,* $E = V$*), or for any sampling sequence* s*, there exist* F *and* s' *such that* $E \mathbin{@} s \Rightarrow F \mathbin{@} s'$*.*

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. □

The *syntactic* distinction between terms and expressions in $\lambda_{\bigcirc}$ is optional in the sense that the grammar does not need to distinguish expressions as a separate non-terminal. On the other hand, the *semantic* distinction, both statically (in the form of two typing judgments) and dynamically (in the form of evaluation and computation judgments) appears to be essential for a clean formulation of our probabilistic language.

$\lambda_{\bigcirc}$ is a conservative extension of a conventional language because terms constitute a conventional language of their own. In essence, term evaluations are always deterministic and we need only terms when writing deterministic programs. As a separate syntactic category, expressions also provide a framework for probabilistic computation that abstracts from the definition of terms. For instance, the addition of a new term construct does not change the definition of expression computations. When programming in $\lambda_{\bigcirc}$, therefore, 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 trying to calculate the probability assigned to each event. If the probability distribution itself is defined in terms of a process of generating samples, we simply translate the definition. 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 [2].

We assume primitive types int and bool, 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. While we do not discuss here type inference or polymorphism, the implementation handles these in the manner familiar from ML. 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 and returns it. 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 an example for each case.

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

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

$\mathit{bernoulli}$ can be thought of as a binary choice construct. It is expressive enough to specify any discrete distribution with finite support. In fact, $\mathit{bernoulli}\ 0.5$ suffices to specify all such probability distributions, since it is capable of simulating a binary choice construct [5].

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:

$$\begin{array}{l} \mathsf{let}\ \mathit{uniform} = \lambda a{:}\mathsf{real}.\, \lambda b{:}\mathsf{real}. \\ \quad \mathsf{prob}\ \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ \mathcal{S}\ \mathsf{in}\ a + x * (b - a) \end{array}$$

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:

$$\begin{array}{l} \mathsf{let}\ \mathit{point\_uniform} = \mathsf{prob}\ \ \mathsf{sample}\ x\ \mathsf{from}\ \mathsf{prob}\ \mathcal{S}\ \mathsf{in} \\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{if}\ x < 0.5\ \mathsf{then}\ 0.0\ \mathsf{else}\ x \end{array}$$

Rich expressiveness

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:

$$\begin{array}{l} \mathsf{let}\ \mathit{binomial} = \lambda p{:}\mathsf{real}.\, \lambda n_0{:}\mathsf{int}. \\ \quad \mathsf{let}\ \mathit{bernoulli}_p = \mathit{bernoulli}\ p\ \mathsf{in} \\ \quad \mathsf{let\ rec}\ \mathit{binomial}_p = \lambda n{:}\mathsf{int}. \\ \qquad \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{prob}\ 0 \\ \qquad \mathsf{else}\ \mathsf{prob}\ \ \mathsf{sample}\ x\ \mathsf{from}\ \mathit{binomial}_p\ (n-1)\ \mathsf{in} \\ \qquad\qquad\qquad\quad \mathsf{sample}\ b\ \mathsf{from}\ \mathit{bernoulli}_p\ \mathsf{in} \\ \qquad\qquad\qquad\quad \mathsf{if}\ b\ \mathsf{then}\ 1 + x\ \mathsf{else}\ x \\ \qquad \mathsf{in} \\ \quad \mathit{binomial}_p\ n_0 \end{array}$$

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

If a probability distribution is defined in terms of a recursive process of generating samples, we can translate the definition into a recursive term. For instance, we encode a geometric distribution with parameter p as follows:

$$
\begin{array}{l}
\mathsf{let}\ \mathit{geometric\_rec} = \lambda p\colon \mathsf{real}.\\
\quad \mathsf{let}\ \mathit{bernoulli}_p = \mathit{bernoulli}\ p\ \mathsf{in}\\
\quad \mathsf{let\ rec}\ \mathit{geometric} =\\
\qquad \mathsf{prob}\ \ \mathsf{sample}\ b\ \mathsf{from}\ \mathit{bernoulli}_p\ \mathsf{in}\\
\qquad\qquad \mathsf{eif}\ b\ \mathsf{then}\ 0\\
\qquad\qquad \mathsf{else}\ \ \mathsf{sample}\ x\ \mathsf{from}\ \mathit{geometric}\ \mathsf{in}\\
\qquad\qquad\qquad 1 + x\\
\quad \mathsf{in}\\
\quad \mathit{geometric}
\end{array}
$$

Note that a geometric distribution has infinite support.

We encode an exponential distribution by using the inverse of its cumulative distribution function as a sampling function, which is known as the *inverse transform method*:

$$\mathsf{let}\ \mathit{exponential}_{1.0} = \mathsf{prob\ sample}\ x\ \mathsf{from}\ \mathcal{S}\ \mathsf{in}\ -\mathsf{log}\ x$$

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

$$
\begin{array}{l}
\mathsf{let}\ \mathit{bernoulli}_{0.5} = \mathit{bernoulli}\ 0.5\\
\mathsf{let}\ \mathit{gaussian\_rejection} = \lambda m\colon \mathsf{real}.\ \lambda \sigma\colon \mathsf{real}.\\
\quad \mathsf{let\ rec}\ \mathit{gaussian} =\\
\qquad \mathsf{prob}\ \ \mathsf{sample}\ y_1\ \mathsf{from}\ \mathit{exponential}_{1.0}\ \mathsf{in}\\
\qquad\qquad \mathsf{sample}\ y_2\ \mathsf{from}\ \mathit{exponential}_{1.0}\ \mathsf{in}\\
\qquad\qquad \mathsf{eif}\ y_2 \geq (y_1 - 1.0)^2/2.0\ \mathsf{then}\\
\qquad\qquad\quad \mathsf{sample}\ b\ \mathsf{from}\ \mathit{bernoulli}_{0.5}\ \mathsf{in}\\
\qquad\qquad\quad \mathsf{if}\ b\ \mathsf{then}\ m + \sigma * y_1\ \mathsf{else}\ m - \sigma * y_1\\
\qquad\qquad \mathsf{else\ unprob}\ \mathit{gaussian}\\
\quad \mathsf{in}\\
\quad \mathit{gaussian}
\end{array}
$$

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 $Prob(x, y) \propto P(x)Q(y)$:

$$
\begin{array}{l}
\mathsf{prob}\ \ \mathsf{sample}\ x\ \mathsf{from}\ M_P\ \mathsf{in}\\
\qquad \mathsf{sample}\ y\ \mathsf{from}\ M_Q\ \mathsf{in}\\
\qquad (x, y)
\end{array}
$$

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 $Prob(x, y) \propto P(x)Q(y|x)$:

$$
\begin{array}{l}
\mathsf{prob}\ \ \mathsf{sample}\ x\ \mathsf{from}\ M_P\ \mathsf{in}\\
\qquad \mathsf{sample}\ y\ \mathsf{from}\ M_Q\ x\ \mathsf{in}\\
\qquad (x, y)
\end{array}
$$

We compute the integration $Prob(y) = \int Q(y|x)P(x)dx$ in a similar way:

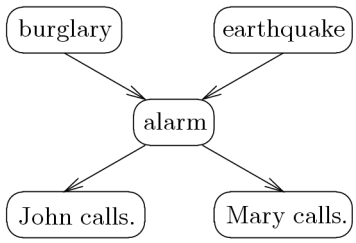
$$\begin{array}{ll} \mathsf{prob} & \mathsf{sample}\ x\ \mathsf{from}\ M_P\ \mathsf{in} \\ & \mathsf{sample}\ y\ \mathsf{from}\ M_{Q|P}\ x\ \mathsf{in} \\ & y \end{array}$$

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 instance, we add two probability distributions over angular values in a straightforward way:

$$\begin{array}{l} \mathsf{let}\ add\_angle = \lambda a_1 : \bigcirc\mathsf{real}.\ \lambda a_2 : \bigcirc\mathsf{real}. \\ \quad \begin{array}{ll} \mathsf{prob} & \mathsf{sample}\ s_1\ \mathsf{from}\ a_1\ \mathsf{in} \\ & \mathsf{sample}\ s_2\ \mathsf{from}\ a_2\ \mathsf{in} \\ & (s_1 + s_2)\ \mathsf{mod}\ (2.0 * \pi) \end{array} \end{array}$$

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 [26]:



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

$$\begin{array}{l} \mathsf{let}\ alarm = \lambda(burglary, earthquake) : \mathsf{bool} \times \mathsf{bool}. \\ \quad \mathsf{if}\ burglary\ \mathsf{then}\ P_{alarm|burglary} \\ \quad \mathsf{else\ if}\ earthquake\ \mathsf{then}\ P_{alarm|\neg burglary \wedge earthquake} \\ \quad \mathsf{else}\ P_{alarm|\neg burglary \wedge \neg earthquake} \\ \mathsf{let}\ john\_calls = \lambda alarm : \mathsf{bool}. \\ \quad \mathsf{if}\ alarm\ \mathsf{then}\ P_{John\_calls|alarm} \\ \quad \mathsf{else}\ P_{John\_calls|\neg alarm} \\ \mathsf{let}\ mary\_calls = \lambda alarm : \mathsf{bool}. \\ \quad \mathsf{if}\ alarm\ \mathsf{then}\ P_{Mary\_calls|alarm} \\ \quad \mathsf{else}\ P_{Mary\_calls|\neg alarm} \end{array}$$

The conditional probabilities $alarm$, $john\_calls$, and $mary\_calls$ do not answer any query on the belief network; they only describe its structure. In order to answer a specific query, we have to implement a corresponding probability distribution. For instance, 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 essen-

tially implements logic sampling [8]:

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

$P_{burglary}$ denotes the probability distribution that a burglary happens, and $P_{earthquake}$ denotes 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 $\sharp$ (the second update equation of the Bayes filter uses it). $P \sharp Q$ 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 and Q to achieve a specialized implementation. For instance, if we have a function p and a constant c such that $p(x) = kP(x) \leq c$ for a certain constant k, we can implement $P \sharp Q$ by the rejection method:

$$
\begin{array}{l}
\mathsf{let}\ bayes\_rejection = \lambda p{:}A \to \mathsf{real}.\ \lambda c{:}\mathsf{real}.\ \lambda Q{:}\bigcirc A. \\
\quad \mathsf{let\ rec}\ bayes = \\
\qquad \mathsf{prob}\ \ \mathsf{sample}\ x\ \mathsf{from}\ Q\ \mathsf{in} \\
\qquad\quad \mathsf{sample}\ u\ \mathsf{from\ prob}\ \mathcal{S}\ \mathsf{in} \\
\qquad\quad \mathsf{eif}\ u < (p\ x)/c\ \mathsf{then}\ x\ \mathsf{else\ unprob}\ bayes \\
\qquad \mathsf{in} \\
\quad bayes
\end{array}
$$

We will see another implementation in Section 6.

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 instance, $exponential_{1.0}$ 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:

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

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

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

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

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

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 instance, 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 directly in terms of probability distributions.

We take a simple approach that appeals to our intuition on the meaning of expressions. We write $E \sim Prob$ 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 $E \sim Prob$, then E denotes a probabilistic computation of generating samples from $Prob$ and we regard $Prob$ as the denotation of $\mathsf{prob}\ E$.

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 $\mathsf{prob}\ \mathcal{S}$. The computation of S proceeds as follows:

$$\mathcal{S}\ @\ R_1^\infty \Rightarrow R_1\ @\ R_2^\infty$$

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:

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

Since R_1 is a random variable from $U(0.0, 1.0]$, the probability of $R_1 \leq p$ 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:

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

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 $Binomial_{p,n}$:

Proposition 5.1. *If* $binomial_p\ n \mapsto^* \mathsf{prob}\ E_{p,n}$*, then* $E_{p,n} \sim Binomial_{p,n}$.

Proof. By induction on n.

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

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

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

By induction hypothesis, $binomial_p\ (n-1)$ generates a sample $x_{p,n-1}$ from $Binomial_{p,n-1}$ after consuming $R_1 \cdots R_{i-1}$ for some i (which is actually n). Since R_i is an independent random variable, $bernoulli_p$ 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

$p * Binomial_{p,n-1}(k-1) + (1.0 - p) * Binomial_{p,n-1}(k) = Binomial_{p,n}(k)$.

Thus we have $E_{p,n} \sim Binomial_{p,n}$. □

As a final example, we show that $geometric\_rec\ p$ denotes a geometric distribution with parameter p. Suppose $geometric \mapsto^*$ prob E and $E \sim Prob$. The computation of E proceeds as follows:

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

The first case happens with probability p and we get $Prob(0) = p$. In the second case, we compute the same expression E with sampling sequence 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 sampling sequence R_2^∞ returns samples from the same probability distribution $Prob$ and we get $Prob(1+k) = (1.0-p) * Prob(k)$. Solving the two equations, we get $Prob(k) = p * (1.0-p)^{k-1}$, 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 method established in simulation theory [2]. An alternative approach would be to develop a denotational semantics. For instance, if we ignore fixed point constructs, it is straightforward to translate expressions into probability measures because probability measures form a monad [6, 25] and expressions already follow a monadic syntax.[2] In practice, however, the translation does not immediately reveal the probability measure corresponding to a given expression and we have to go through essentially the same analysis as in the above approach. Ultimately we have to invert a sampling function represented by a given expression (because an event is assigned a probability proportional to the size of its inverse image under the sampling function), but this is difficult to do in a mechanical way in the presence of various operators. Therefore it seems to be reasonable to analyze each expression individually as demonstrated in this section.

6 Approximate Computation in $\lambda_\bigcirc$

We have explored both how to encode probability distributions in $\lambda_\bigcirc$ and how to interpret $\lambda_\bigcirc$ in terms of probability distributions. In this section, we discuss another important aspect of probabilistic languages: reasoning about probability distributions.

The expressive power of a probabilistic language is an important factor affecting its practicality. Another important factor is its support for reasoning about probability distributions to determine their properties. In other words, it is important not only to be able to encode various probability distributions but also to be able to determine their properties such as means, variances, and probabilities of specific events. Unfortunately $\lambda_\bigcirc$ does not support precise reasoning about probability distributions. That is, it does not permit a precise implementation of queries on probability distributions. Intuitively we must be able to calculate probabilities of specific events, but this is essentially inverting sampling functions.

Given that we cannot hope for precise reasoning in $\lambda_\bigcirc$, we choose to support approximate reasoning by the Monte Carlo method [13]. It approximately answers a query on a probability distribution by generating a large number of samples and then analyzing them. For instance, in the belief network example in Section 4, we can approximate $p_{Mary\_calls|John\_calls}$ by generating a large number of samples and counting the number

[2] In the presence of fixed point constructs, expressions should be translated into a domain-theoretic structure because of recursive equations. While the work by Jones [10] suggests that such a structure could be constructed from a domain-theoretic model of real numbers, we have not investigated in this direction.

of True's. Although the Monte Carlo method gives only an approximate answer, its accuracy improves with the number of samples. Moreover it can be applied to all kinds of probability distributions and is therefore particularly suitable for $\lambda_{\bigcirc}$.

In this section, we apply the Monte Carlo method to an implementation of the expectation query. We also show how to exploit the Monte Carlo method in implementing the Bayes operation. Then we briefly describe our implementation of $\lambda_{\bigcirc}$.

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 instance, the mean of a probability distribution over real numbers is the expectation of an identity function.

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:

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

$$\frac{\Gamma \vdash M_f : A \to \mathsf{real} \quad \Gamma \vdash M_P : \bigcirc A}{\Gamma \vdash \mathsf{expectation}\ M_f\ M_P : \mathsf{real}}\ \mathsf{Exp} \qquad \frac{\begin{array}{c} M_f \mapsto^* f \quad M_P \mapsto^* \mathsf{prob}\ E_P \\ E_P\ @\ s_i \Rightarrow^* V_i\ @\ s_i' \quad f\ V_i \mapsto^* v_i \quad 1 \le i \le n \end{array}}{\mathsf{expectation}\ M_f\ M_P \mapsto_{\mathsf{R}} \frac{\sum_i v_i}{n}}\ \mathsf{Exp_R}$$

The rule $\mathsf{Exp_R}$ 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 runtime variable n specifies the number of samples to be generated from P. The runtime system initializes sampling sequence s_i to generate sample V_i.

A problem with the above definition is that although expectation is a term construct, its reduction is probabilistic because of sampling sequence s_i in the rule $\mathsf{Exp_R}$. This violates the principle that a term evaluation is always deterministic, and now the same term may evaluate to different values if it contains expectation. In practice, however, this is acceptable because $\lambda_{\bigcirc}$ is intended to be embedded in Objective CAML in which side-effects are already allowed for terms. Besides, mathematically the expectation of a function with respect to a probability distribution is always unique (if it exists).[3]

Now we can calculate $p_{Mary\_calls|John\_calls}$ as:

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

6.2 Bayes operation

The previous implementation of the Bayes operation $P \sharp Q$ assumes that we have a function p and a constant c such that $p(x) = kP(x) \le c$ 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 [13], allows us to dispense with c by approximating

[3] We define expectation as a term construct only for pragmatic reasons. For instance, examples in Section 7 become much more complicated if expectation is defined as an expression construct.

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:

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

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

$$\frac{\Gamma \vdash M_p : A \to \mathsf{real} \quad \Gamma \vdash M_Q : \bigcirc A}{\Gamma \vdash \mathsf{bayes}\ M_p\ M_Q : \bigcirc A}\ \mathsf{Bayes} \qquad \frac{\Gamma \vdash V_i : A \quad \Gamma \vdash w_i : \mathsf{real} \quad 1 \leq i \leq n}{\Gamma \vdash \mathsf{importance}\ \{(V_i, w_i) | 1 \leq i \leq n\} \div A}\ \mathsf{Imp}$$

$$\frac{\begin{array}{c} M_p \mapsto^* p \quad M_Q \mapsto^* \mathsf{prob}\ E_Q \\ E_Q\ @\ s_i \Rightarrow^* V_i\ @\ s_i' \quad p\ V_i \mapsto^* w_i \quad 1 \leq i \leq n \end{array}}{\mathsf{bayes}\ M_p\ M_Q \mapsto_{\mathsf{R}} \mathsf{prob}\ \mathsf{importance}\ \{(V_i, w_i) | 1 \leq i \leq n\}}\ \mathsf{Bayes_R}$$

$$\frac{\frac{\sum_{i=1}^{k-1} w_i}{S} < r \leq \frac{\sum_{i=1}^{k} w_i}{S} \quad \mathit{where} \quad S = \sum_{i=1}^{n} w_i}{\mathsf{importance}\ \{(V_i, w_i) | 1 \leq i \leq n\}\ @\ rs \Rightarrow_{\mathsf{R}} V_k\ @\ s}\ \mathsf{Imp_R}$$

The rule $\mathsf{Bayes_R}$ approximates Q with n samples $V_1, \cdots, V_n$, where n is a runtime variable as in the rule $\mathsf{Exp_R}$. Then it applies p to each sample V_i to calculates its weight w_i and creates a set $\{(V_i, w_i) | 1 \leq i \leq n\}$ of weighted samples as an argument to importance. The rule $\mathsf{Imp_R}$ 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.

As with expectation, we decide to define bayes as a term construct despite the fact that its reduction is probabilistic. The decision also conforms to our intuition that mathematically the result of the Bayes operation between two probability distributions is always unique.

6.3 Implementation of $\lambda_\bigcirc$

We have implemented $\lambda_\bigcirc$ by extending the syntax of Objective CAML. The runtime system uses a global random number generator for all sampling sequences. Hence it generates fresh random numbers whenever it needs to compute sampling expressions, without explicitly initializing sampling sequences. The runtime system also allows the programmer to change the runtime variable n in the rules $\mathsf{Exp_R}$ and $\mathsf{Bayes_R}$, both of which invoke expression computations during term evaluations. Thus the programmer can control the accuracy in approximating probability distributions.

7 Applications

In this section, we present three applications of $\lambda_\bigcirc$ in robotics: robot localization, people tracking, and robotic mapping. The goal is to estimate the state of a robot from sensor readings, where the definition of state differs in each case. In order to cope with uncertainty in sensor readings (due to limitations of sensors and noises from the environment), we estimate the state with a probability distribution. We use a Bayes filter as a framework for updating the probability distribution.

There are 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 is represented as an odometry reading which returns the pose (*i.e.*, position and orientation) of the robot relative to its initial pose. A measurement includes range readings which return distances to objects at certain angles (see Figure 6 in Appendix).

We first consider robot localization, since it directly implements update equations (1) and (2) in Section 2.

7.1 Robot localization

Robot localization [28] 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 difficult than pose tracking).

We consider robot localization under the assumption that the environment is static. This assumption allows us to use a Bayes filter over 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 update equations (1) and (2) with the following interpretation:

- $\mathcal{A}(s|a, s')$ is the probability that the robot moves to pose s after taking action a in 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 measurement m_s for the pose s, and the difference between m and m_s is 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 can implement update equations (1) and (2) as follows:

$$\left.\begin{array}{ll} \mathsf{let}\ Bel_{new} = \mathsf{prob} & \mathsf{sample}\ s'\ \mathsf{from}\ Bel\ \mathsf{in} \\ & \mathsf{sample}\ s\ \mathsf{from}\ M_{\mathcal{A}}\ (a, s')\ \mathsf{in} \\ & s \end{array}\right\} (1)$$

$$\left.\begin{array}{ll} \mathsf{let}\ Bel_{new} = \mathsf{bayes}\ (M_{\mathcal{P}}\ m)\ Bel & \end{array}\right\} (2)$$

Now we can implement pose tracking or global localization by specifying an initial probability distribution of robot pose. In the case of pose tracking, it is usually a point-mass distribution or a Gaussian distribution; in the case of global localization, it is usually a uniform distribution over the open space in the map.

7.2 People tracking

People tracking [20] is an extension of robot localization in that it estimates not only the robot pose but also the positions of people (or unmapped objects). As in robot localization, the robot can take an action to change its pose. Unlike in robot localization, however, the robot must categorize sensor readings in a measurement by deciding whether they are caused by objects in the map or by 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 the 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. While 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 multiple incorrect hypotheses that lead to incorrect interpretation of those sensor readings. This means that during global localization, there exists a dependence between the robot pose and the positions of people, which is not captured by $Bel(s, \vec{u})$.

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

Figure 3: Equations used in people tracking. (3) and (4) for the Bayes filter computing $Bel(s, P_s(\vec{u}))$. (5) for decomposing the action model. (6) and (7) for the inner Bayes filter computing $P_s(\vec{u})$.

$$
\begin{array}{l}
\mathsf{let}\ Bel_{new} = \mathsf{prob}\ \ \mathsf{sample}\ (s', P_{s'}(\vec{u'}))\ \mathsf{from}\ Bel\ \mathsf{in} \\
\qquad\qquad \mathsf{sample}\ s\ \mathsf{from}\ M_{\mathcal{A}_{\mathsf{robot}}}\ (a, m, s', P_{s'}(\vec{u'}))\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ P_s(\vec{u}) = \mathsf{prob}\ \ \mathsf{sample}\ \vec{u'}\ \mathsf{from}\ P_{s'}(\vec{u'})\ \mathsf{in} \\
\qquad\qquad\qquad\qquad \mathsf{sample}\ \vec{u}\ \mathsf{from}\ M_{\mathcal{A}_{\mathsf{people}}}\ (a, \vec{u'}, s, s')\ \mathsf{in} \\
\qquad\qquad\qquad\qquad \vec{u} \\
\qquad\qquad \mathsf{in} \\
\qquad\qquad \mathsf{let}\ P_s(\vec{u}) = \mathsf{bayes}\ (M_{\mathcal{P}_{\mathsf{people}}}\ m\ s)\ P_s(\vec{u})\ \mathsf{in} \\
\qquad\qquad (s, P_s(\vec{u})) \\
\mathsf{let}\ Bel_{new} = \mathsf{bayes}\ \lambda(s, P_s(\vec{u})) \ldots (\mathsf{expectation}\ (M_{\mathcal{P}_{\mathsf{people}}}\ m\ s)\ P_s(\vec{u}))\ Bel
\end{array}
$$

(6), (5), (7), (3), (4)

Figure 4: Implementation of people tracking in $\lambda_{\bigcirc}$. Numbers on the right-hand side show corresponding equations in Figure 3.

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 the 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 (3) and (4) in Figure 3 (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 $\mathcal{A}(s, P_s(\vec{u})|a, m, s', P_{s'}(\vec{u'}))$ requires us to generate $s, P_s(\vec{u})$ from $s', P_{s'}(\vec{u'})$ utilizing action a and measurement m. We generate first s and next $P_s(\vec{u})$ according to equation (5) in Figure 3. We write the first $Prob$ in equation (5) as $\mathcal{A}_{\mathsf{robot}}(s|a, m, s', P_{s'}(\vec{u'}))$. The second $Prob$ in equation (5) indicates that we have to 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 (6) and (7) in Figure 3. We write $Prob$ in equation (6) as $\mathcal{A}_{\mathsf{people}}(\vec{u}|a, \vec{u'}, s, s')$; we simplify $Prob$ in equation (7) into $Prob(m|\vec{u}, s)$ because m does not depend on s' given s, and write it as $\mathcal{P}_{\mathsf{people}}(m|\vec{u}, s)$.

Figure 4 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

$$
\begin{array}{l}
\left.\begin{array}{l}
\text{let } Bel_{new} = \\
\quad \text{prob} \;\; \text{sample } (s', P_{s'}(\vec{u'})) \text{ from } Bel \text{ in} \\
\qquad \left.\begin{array}{l}
\text{sample } s \text{ from} \\
\qquad \text{bayes } \lambda s\text{:\_. (expectation } (M_{\mathcal{P}_{\text{landmark}}}\; m\; s)\; P_{s'}(\vec{u'}))\; (M_{\mathcal{A}_{\text{robot}}}\; (a, s')) \text{ in}
\end{array}\right\} (8) \\
\qquad \left.\begin{array}{l}
\text{let } P_s(\vec{u}) = \text{bayes } (M_{\mathcal{P}_{\text{landmark}}}\; m\; s)\; P_s(\vec{u}) \text{ in} \\
(s, P_s(\vec{u}))
\end{array}\right\} (7)
\end{array}\right\} (5) \\
\text{let } Bel_{new} = \text{bayes } \lambda(s, P_s(\vec{u}))\text{:\_. (expectation } (M_{\mathcal{P}_{\text{landmark}}}\; m\; s)\; P_s(\vec{u}))\; Bel
\end{array}\right\} (3) \quad \} \; (4)
$$

Figure 5: Implementation of robotic mapping in $\lambda_{\bigcirc}$. Compared with the implementation in Figure 4, it omits equation (6) and uses equation (8).

a constant κ. In implementing update equation (4), we exploit the fact that $\mathcal{P}(m|s, P_s(\vec{u}))$ is the expectation of a function $g(\vec{u}) = \mathcal{P}_{\text{people}}(m|\vec{u}, s)$ with respect to $P_s(\vec{u})$:

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

We can further simplify the models used in the update equations. For example, we can use $\mathcal{A}_{\text{robot}}(s|a, s')$ instead of $\mathcal{A}_{\text{robot}}(s|a, m, s', P_{s'}(\vec{u'}))$ as in robot localization. In our implementation, we use $\mathcal{A}_{\text{people}}(\vec{u}|\vec{u'})$ on the assumption that the positions of people are not affected by the robot pose.

7.3 Robotic mapping

Robotic mapping [30] 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 must simultaneously localize itself as it explores the environment so that it can correct and align the local snapshots to construct a global map. For this reason, robotic mapping is also referred to as simultaneous localization and mapping (or SLAM). It is one of the most difficult problems in robotics, and is under active research.

We assume that the environment consists of an unknown number of stationary landmarks. Then the goal is to estimate the positions of landmarks as well as the robot pose. The key observation is that we can 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 3. 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 can skip update equation (6) in implementing update equation (3). $\mathcal{A}_{\text{robot}}$ in equation (5) can use $\mathcal{P}_{\text{landmark}}(m|\vec{u'}, s)$ to test the likelihood of each new robot pose s with respect to old positions $\vec{u'}$ of landmarks, as in FastSLAM 2.0 [19]:

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

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

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

Figure 5 shows the implementation of robotic mapping in $\lambda_{\bigcirc}$. $M_{\mathcal{A}_{\mathsf{robot}}}$ and $M_{\mathcal{P}_{\mathsf{landmark}}}$ are interpreted in the same way as in people tracking. Since landmarks are stationary, we no longer need $M_{\mathcal{A}_{\mathsf{landmark}}}$.

7.4 Experimental results

We have implemented the above three systems in $\lambda_{\bigcirc}$. To test the robot localizer and the people tracker, we use a mobile robot Nomad XR4000 in Wean Hall at Carnegie Mellon University. We use CARMEN [18] for controlling the robot and collecting sensor readings. To test the mapper, we use the a data set collected with an outdoor vehicle in Victoria Park, Sydney [1]. All the systems run on Pentium III 500Mhz with 384 MBytes memory.

We test the robot localizer for global localization with 8 runs in Wean Hall (each run takes a different path). For an initial probability distribution of robot pose, we use a uniform distribution over the open space in the map. In a test experiment, it succeeds to localize the robot on 5 runs and fails on 3 runs (see Figures 7 and 8 in Appendix for a successful run). As a comparison, the CARMEN robot localizer, which uses particle filters, succeeds on 3 runs and fails on 5 runs.

The people tracker uses the implementation in Figure 4 during global localization, but once it succeeds to localize the robot and starts pose tracking, it maintains an independent probability distribution for each person in sight (because there is no longer a dependence between the robot pose and the positions of people). Figures 9 and 10 in Appendix show some results from the people tracker.

We test the mapper with a data set in which the vehicle moves approximately 323.42 meters (according to the odometry readings) in 128.8 seconds. Since the vehicle is driving over uneven terrain, raw odometry readings are noisy and do not reflect the true path of the vehicle, in particular when the vehicle follows a loop (see Figure 11). The mapper successfully closes the loop, building a map of the landmarks around the path (see Figure 12). The experiment takes 145.89 seconds.

Our finding is that the benefit of implementing probabilistic computations in $\lambda_{\bigcirc}$, such as readability and conciseness of code, outweighs its disadvantage in speed. As a comparison (although not particularly meaningful), our robot localizer is 1349 lines long (868 lines of Objective CAML code and 481 lines of C code), and the CARMEN robot localizer, written in C, is 3397 lines long. 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).

8 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 [27] 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 [11] present a first order functional language with a coin toss construct $\mathsf{flip}(p)$. Pfeffer [22] 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 [7] 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 [24] 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 [10] presents a metalanguage with a binary choice construct e_1 or$_p$ e_2. Its operational semantics uses a judgment $e \Rightarrow \sum p_i v_i$. Mogensen [15] presents a language for specifying die-rolls. Its denotation semantics (called *probability semantics*) is formulated in a similar style, directly 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). Such a probability distribution is, however, difficult to represent accurately because of an infinite number of elements in its support. For this reason, 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 we can omit some elements in the enumeration.

There are a few probabilistic languages supporting continuous distributions. Kozen [12] investigates the semantics of probabilistic while programs. A random assignment $x :=$ 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 [29] 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 [21] 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 [25]. They present a stochastic lambda calculus (with a binary choice construct choose p e_1 e_2) whose denotational semantics is based upon the monad of probability measures, or the probability monad [6]. 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 directly to sampling functions; then we interpret it in terms of probability distributions.

9 Conclusion and Future Work

We have presented a probabilistic language $\lambda_{\bigcirc}$ whose mathematical basis is sampling functions. $\lambda_{\bigcirc}$ supports all kinds of probability distributions without drawing a syntactic or semantic distinction. We have demonstrated the practicality of $\lambda_{\bigcirc}$ with three applications in robotics. 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 [11, 22] and the limiting process in [7].

$\lambda_{\bigcirc}$ does not support precise reasoning about probability distributions. Note, however, that this is not an inherent weakness 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 is impossible in general. In other words, if $\lambda_{\bigcirc}$ supported precise reasoning, it could support only a small number of probability distributions and operations on them.

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 all kinds of probability distributions are used or precise reasoning is unnecessary. Robotics is a good example, since all kinds of probability 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.

We are investigating how to generate a large number of samples quickly, which is important for improving accuracy of approximate reasoning in $\lambda_{\bigcirc}$. For instance, instead of computing a given expression repeatedly (as in the current implementation of $\lambda_{\bigcirc}$), we could run through it only once by performing multiple, either independent or correlated, computations simultaneously.

References

[1] `http://www.acfr.usyd.edu.au/homepages/academic/enebot/dataset.htm`. Australian Centre for Field Robotics, The University of Sydney.

[2] P. Bratley, B. Fox, and L. Schrage. *A guide to simulation*. Springer Verlag, 2nd edition, 1996.

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

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

[5] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.

[6] M. Giry. A categorical approach to probability theory. In B. Banaschewski, editor, *Categorical Aspects of Topology and Analysis*, volume 915 of *Lecture Notes In Mathematics*, pages 68–85. Springer Verlag, 1981.

[7] V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 189–202. ACM Press, 1999.

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

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

[10] C. Jones. *Probabilistic Non-Determinism*. PhD thesis, Department of Computer Science, University of Edinburgh, 1990.

[11] D. Koller, D. McAllester, and A. Pfeffer. 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)*, pages 740–747. AAAI Press, 1997.

[12] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

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

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

[15] T. Mogensen. Roll: A language for specifying die-rolls. In V. Dahl and P. Wadler, editors, *PADL 2003*, volume 2562 of *LNCS*, pages 145–159. Springer, 2002.

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

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

[18] M. Montemerlo, N. Roy, and S. Thrun. CARMEN: Carnegie mellon robot navigation toolkit. `http://www.cs.cmu.edu/~carmen/`.

[19] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *Proceedings of the seventeenth International Joint Conference on Artificial Intelligence (IJCAI-03)*. Morgan Kaufmann Publishers, Inc., 2003.

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

[21] S. Park. A calculus for probabilistic languages. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 38–49. ACM Press, 2003.

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

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

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

[25] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 154–165. ACM Press, 2002.

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

[27] N. Saheb-Djahromi. Probabilistic LCF. In J. Winkowski, editor, *Proceedings of the 7th Symposium on Mathematical Foundations of Computer Science*, volume 64 of *LNCS*, pages 442–451. Springer, 1978.

[28] S. Thrun. Probabilistic algorithms in robotics. *AI Magazine*, 21(4):93–109, 2000.

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

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

[31] G. Welch and G. Bishop. An introduction to the kalman filter. Technical Report TR95-041, Department of Computer Science, University of North Carolina - Chapel Hill, 1995.

Appendix

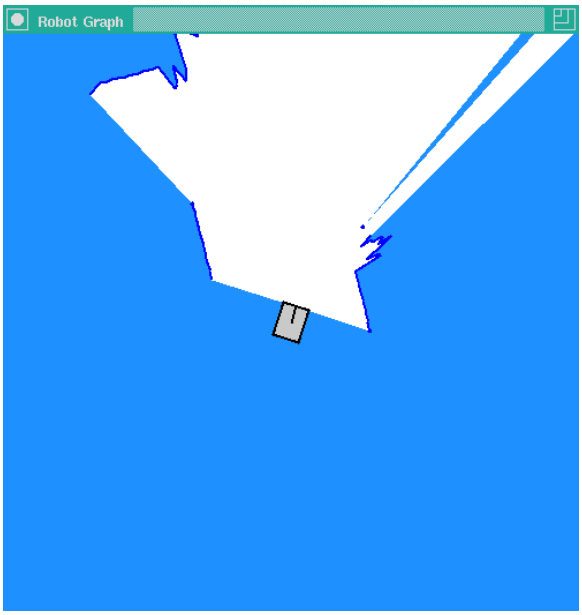


Figure 6: Range readings produced by laser range finders in a mobile robot Nomad XR4000. The robot is shown in the center. An occluded region is colored in light blue.

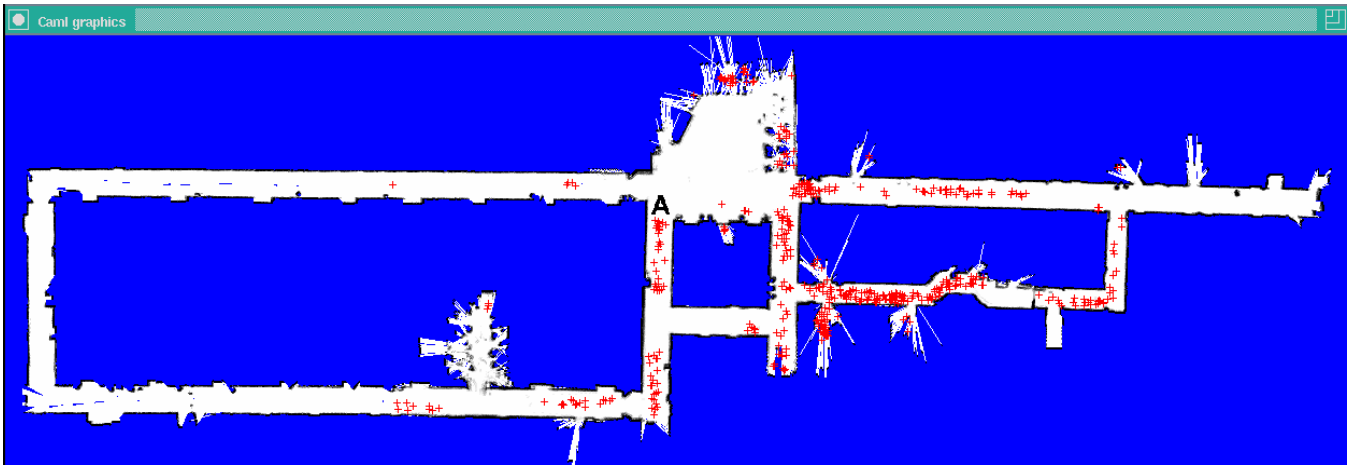


Figure 7: Probability distribution of robot pose after processing the first batch of range readings in Figure 6. Red crosses represent samples generated from the probability distribution. The open space in the map is colored in white. We use 500 samples. The robot starts right below character A, but there are relatively few samples around the true position of the robot.

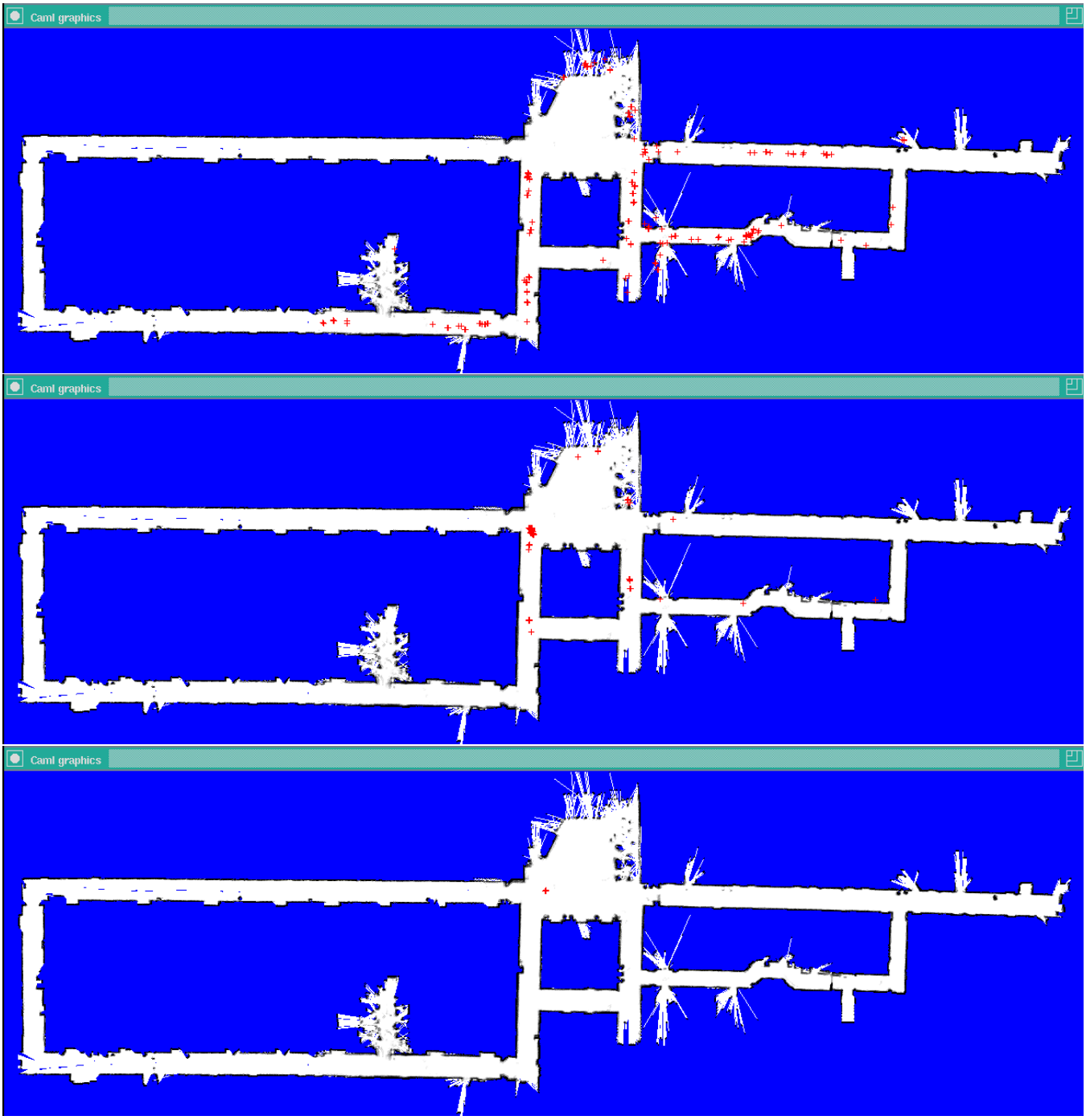


Figure 8: Progress of a real-time robot localization run that continues with the probability distribution in Figure 7. The first two pictures show that the robot localizer is still performing global localization. The last picture shows that the robot localizer has started pose tracking.

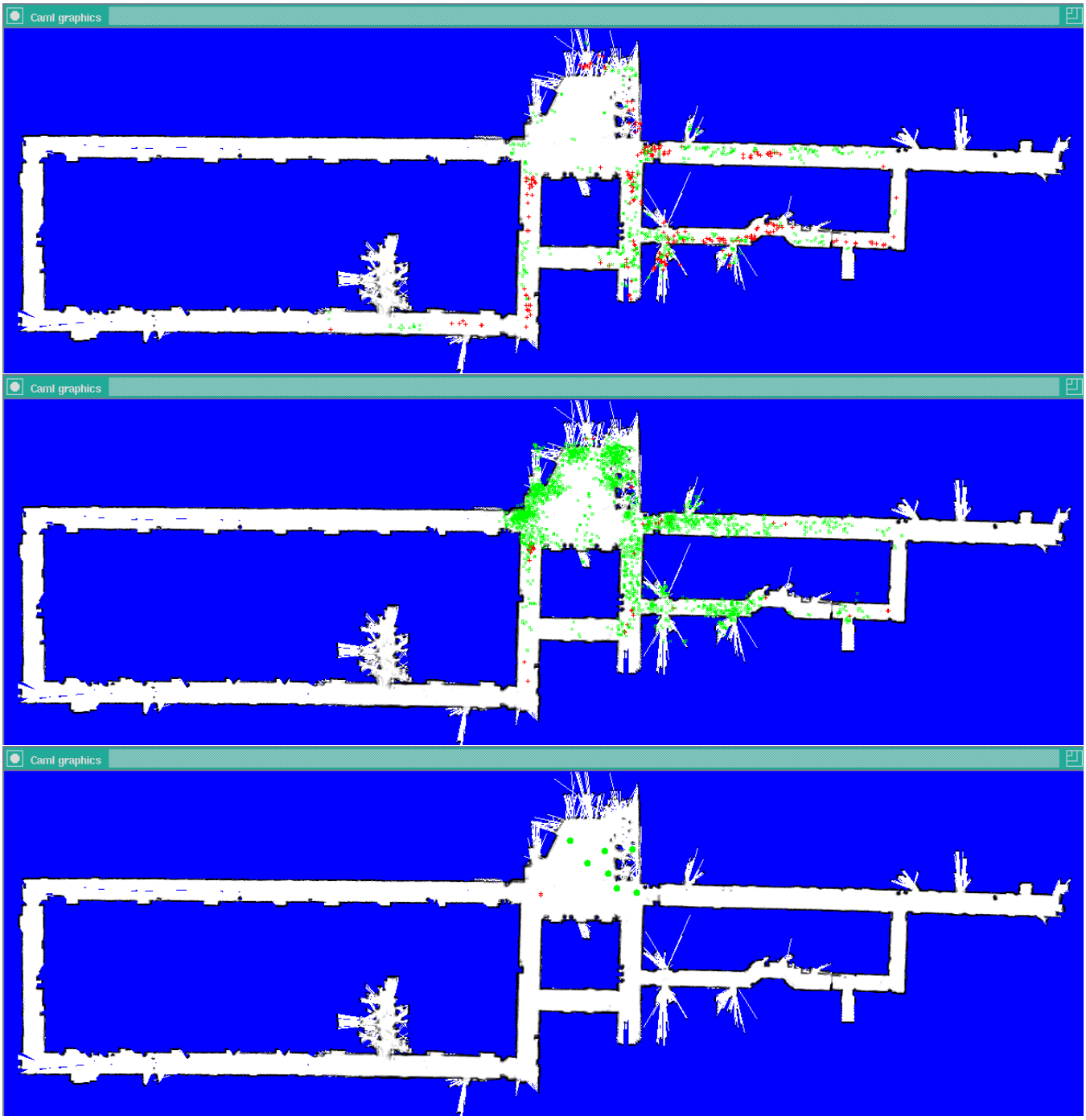


Figure 9: Progress of a real-time people tracking run that uses the same sequence of sensor readings as the previous robot localization run. The first picture is taken after processing the first batch of range readings in Figure 6; red crosses correspond to robot poses and green diagonal crosses correspond to positions of people. The second picture shows that the people tracker is still performing global localization. The last picture shows that the people tracker has started pose tracking; the position of each person in sight is indicated by a green dot.

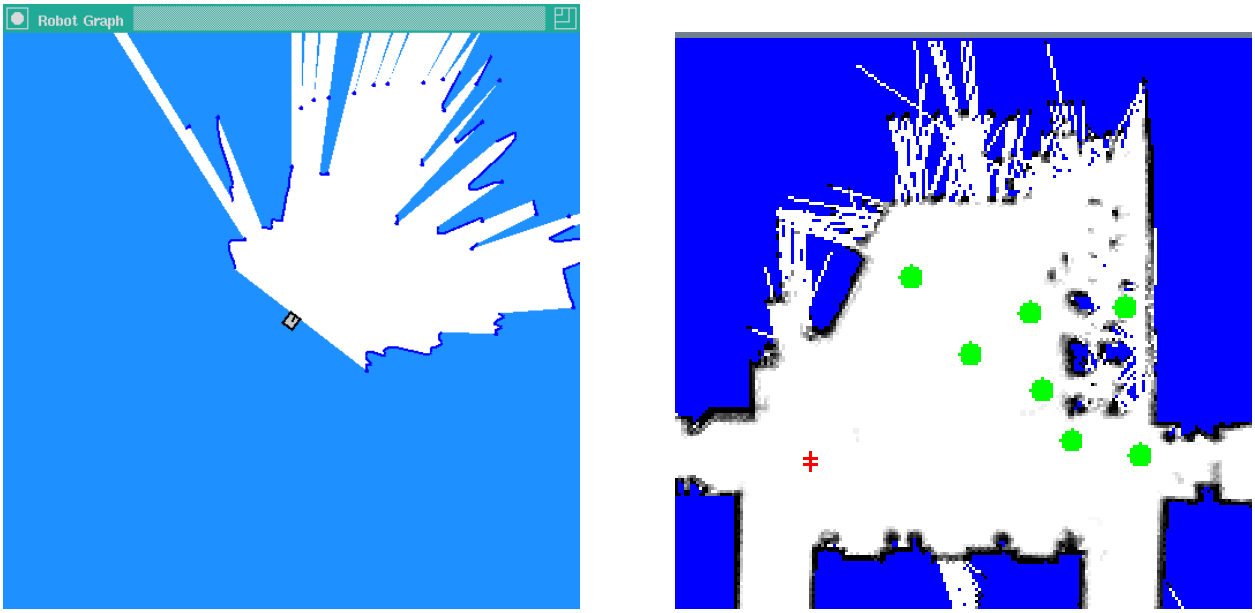


Figure 10: Range readings when the third picture in Figure 9 is taken. The right picture shows a magnified view of the area around the robot. A person may be occluded by another person, so green dots do not always reflect the movement of people instantly.

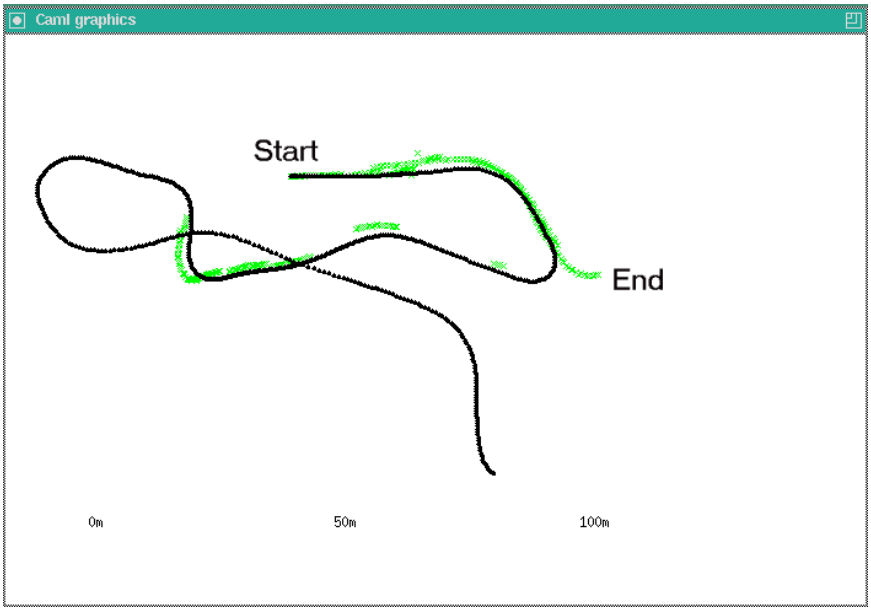


Figure 11: Raw odometry readings in the robotic mapping experiment. Green crosses represent the true positions of the vehicle measured by a GPS sensor. The GPS sensor readings are available only for part of the entire traverse. The odometry readings do not follow the true path of the vehicle.

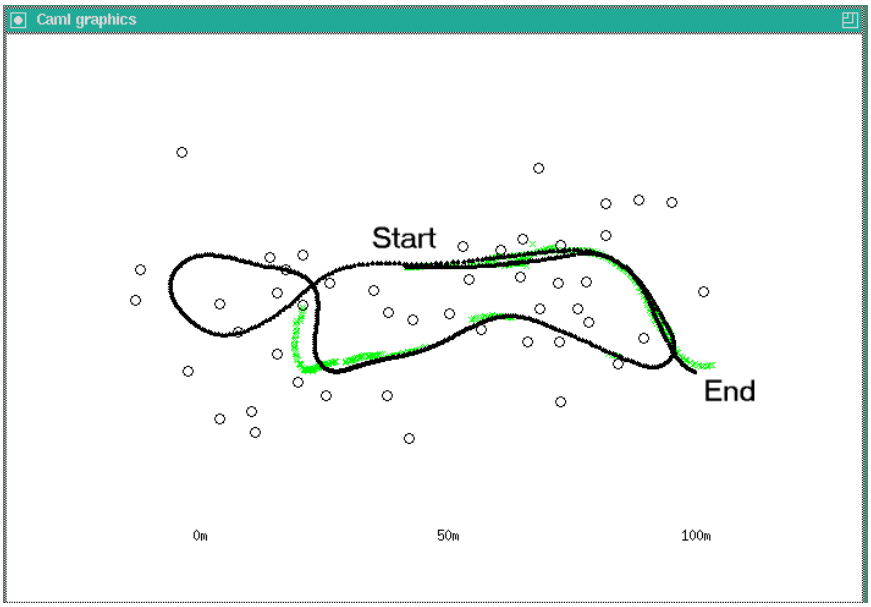


Figure 12: Result of the robotic mapping experiment. The mapper successfully closes the loop. The circles represent landmark positions (mean of their probability distributions).