A Calculus for Probabilistic Languages

Sungwoo Park
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
gla@cs.cmu.edu

ABSTRACT

As probabilistic computation plays an increasing role in diverse fields in computer science, researchers have designed new languages to facilitate the development of probabilistic programs. In this paper, we develop a probabilistic calculus by extending the traditional lambda calculus. In our calculus, every expression denotes a probability distribution yet evaluates to a regular value. The most notable feature of our calculus is that it is founded upon sampling functions, which map the unit interval to probability domains. As a consequence, we achieve a unified representation scheme for all types of probability distributions. In order to support an efficient implementation of the calculus, we also develop a refinement type system which is capable of distinguishing expressions denoting regular values from expressions denoting probability distributions. We use a novel formulation of the intuitionistic modal logic S4 with an intersection connective in the refinement type system. We present preliminary evidence that a probabilistic language based upon our calculus is viable in applications involving massive probabilistic computation.

Categories and Subject Descriptors

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

General Terms

Languages

Keywords

Probabilistic language, Probabilistic calculus, Refinement type system

1. INTRODUCTION

As probabilistic computation is regarded as a suitable model of computation and plays an increasing role in diverse fields in computer science, researchers have designed new languages to facilitate the development of *probabilistic programs*, which model probabilistic computation. An example is an imperative language *CES* [19], which is an extension to C++ with probabilistic data types. The language is designed specifically for *probabilistic robotics* [18] and successfully employed in implementing typical robot control programs compactly. For instance, Thrun [19] implements a program for controlling a mail-delivery robot through gestures with only 137 lines of CES code, which is a reduction by two orders of magnitude in code size compared to a previous implementation in C. We refer to such a language as a *probabilistic language* in the broad sense that it incorporates probabilistic computation as an inherent component.

Our goal is to develop a calculus for probabilistic languages, or a *probabilistic calculus*, by extending the traditional lambda calculus with the emphasis that it targets applications involving massive computation on continuous probability distributions (such as arising in probabilistic robotics) as well as discrete probability distributions. Previous work on probabilistic languages does not seem to be suitable for our goal because it supports only discrete probability distributions or its semantics is not well formulated. We also want to draw on type theory to provide programmers with an informative type system and support an efficient implementation of the runtime system.

1.1 Design Criteria

We find that the design of a probabilistic calculus is guided by four fundamental criteria, which are independent of each other.

Value space. There are two approaches depending on which of regular values and probability distributions play a dominant role in the formulation of the calculus. In one approach, every expression denotes a regular value, and a probability distribution is represented as a regular value of a special type. For instance, a regular integer is given type int whereas an integer distribution is given type $\circ int$. In the dual approach, every expression denotes a probability distribution, and a regular value is represented as a special form of probability distribution, namely, a Dirac distribution.

Operational behavior. There are two approaches depending on how the evaluation of an expression denoting a probability distribution proceeds. If the expression evaluates to a regular value, that is, a sample from the probability distribution, an infinite number of evaluations reveals a unique pattern of samples dictated by the probability distribution. On the other hand, if the expression evaluates to the probability distribution itself, a single evaluation reveals the pattern directly or indirectly.

Reduction strategy. In a probabilistic calculus, the difference between call-by-value reduction and call-by-name reduction becomes more pronounced than in the traditional lambda calculus. We define call-by-value reduction in such a way that given an expression denoting a probability distribution, a lambda abstraction first extracts a regular value from its argument, which is then used throughout the evaluation of its body. In this case, a lambda abstraction is mathematically equivalent to a conditional probability, and consequently an application may be thought of as an integration between a conditional probability and a probability distribution.[1] In contrast, call-by-name reduction enables a lambda abstraction to manipulate probability distributions directly without extracting regular values from its argument. For instance, a lambda abstraction $\lambda x.x - x$ considers the two occurrences of x in its body to be independent of each other. Thus it denotes a pure function on probability distributions and is not equivalent to a conditional probability.

Mathematical basis. Since a probability distribution can be specified in various ways, we need to choose a class of mathematical objects to be used as the source of probability distributions. For instance, a probability mass function specifies a unique discrete probability distribution, and thus we can introduce a primitive construct based upon the notion of probability mass function. The decision here is significant because it affects not only the syntax but also the expressive power of the calculus: certain probability distributions may be hard or impossible to specify in terms of the primitive construct provided by the calculus.

1.2 Our Calculus

In this paper, we investigate a probabilistic calculus in which every expression denotes a probability distribution yet evaluates to a regular value. Uniform treatment of expressions leads to a concise type system, and the operational semantics remains simple because it does not need to accumulate all the probability distributions encountered during an evaluation. The calculus supports both call-by-value reduction and call-by-name reduction, which serve different purposes: call-by-value reduction supports conditional probabilities, and call-by-name reduction reduction supports pure functions on probability distributions.

The most notable feature of the calculus is that it is founded upon *sampling functions* to specify probability distributions. A sampling function f maps the unit interval $(0.0, 1.0]$ to a probability domain. By generating a random number from the unit interval according to a uniform distribution and then applying f, we obtain a sample from the probability distribution specified by f. Thus a sampling function specifies a probability distribution by answering *"how can we generate samples according to the probability distribution?"*.

As the syntax for sampling functions, we adopt a *sampling construct* $\gamma.e$, where γ is a formal parameter and e is the body of a sampling function. The evaluation of $\gamma.e$ proceeds by replacing all free occurrences of γ in e by a random number, which is automatically generated by the runtime system.

[1]Throughout this paper, an integration between a conditional probability C and a probability distribution P means an operation returning a probability distribution Q such that $Q(y) = \int_x C(y|x)P(x)dx$.

Since the calculus does not distinguish regular values from probability distributions, a naive type system infers only the probability domain for a given expression. This may obstruct an efficient implementation of the operational semantics because regular values are used extensively even in probabilistic programs and, in general, it requires more information to represent a probability distribution than a regular value on an arbitrary probability domain. Therefore we develop a *refinement type system* in order to statically distinguish expressions denoting regular values from expressions denoting probability distributions. The basic idea is simple: if an expression denoting a probability distribution on a certain probability domain is given type σ, then any expression denoting a regular value on the same probability domain can be given type $\Box\sigma$, which is a subtype of σ. Hence a subtyping principle $\Box\sigma \leq \sigma$ guides the development of the refinement type system.

We first formulate the subtyping relation $\leq$ in the customary way as a binary relation between types. Although provability in the subtyping system is decidable, it is not orthogonal, and various properties of $\leq$ are hard to prove. Therefore we revise it into a logical form to obtain a *structural subtyping system*, which is presented in the style of Gentzen's sequent calculus. The structural subtyping system does not have any distributivity rule, and admissibility of the cut rules shows that we do not need transitivity rules either. Since it is orthogonal and easy to reason about, we can prove important properties of the subtyping relation in a straightforward way. With the structural subtyping system, we formulate the refinement type system and prove its type preservation property. Finally we prove the soundness of the refinement type system: if an expression can be given a type $\Box\sigma$, its evaluation does not consume any random number and thus it denotes a regular value.

In our attempt to prove the equivalence between the two subtyping systems, we discover that our structural subtyping system is indeed a new formulation of the intuitionistic modal logic S4 with an intersection connective. We believe that the new formulation is better than the typical axiomatic characterization of the same logic because it is orthogonal and easier to reason about.

We have preliminary evidence that a probabilistic language based upon our calculus is viable in applications involving massive probabilistic computation over continuous probability domains. We have implemented a robot localization program which employs sampling functions to represent probability distributions. The program has been tested on a real robot.

1.3 Organization of the Paper

In Section 2, we present our calculus λ_γ with a simple type system $\vdash_\gamma$. We also demonstrate how to express various kinds of probability distributions in λ_γ. The discussion of its implementation issues leads to the development the refinement type system $\vdash_\leq$ in Section 3. First we formulate the two subtyping systems and prove their equivalence. We then prove the type preservation property for $\vdash_\leq$ and its soundness. Section 4 discusses previous work on probabilistic languages. Section 5 discusses strengths and weaknesses of λ_γ. It also discusses how to exploit $\vdash_\leq$ in implementing λ_γ. Section 6 presents preliminary results on the practicality of λ_γ and then concludes with future work.

$$\begin{array}{llll}
\text{type} & \tau & ::= & b \mid \tau \rightarrow \tau \mid \tau \Rightarrow \tau \\
\text{base type} & b & ::= & \mathsf{real} \\
\text{expression} & e & ::= & x \mid \lambda x{:}\tau.e \mid e\ e \mid \textstyle\int x{:}\tau.e \mid e \circ e \mid \\
 & & & \gamma \mid \gamma.e \mid u \mid \mathsf{fix}\ x{:}\tau.e \\
\text{value} & v & ::= & \lambda x{:}\tau.e \mid \textstyle\int x{:}\tau.e \mid u \\
\text{random number} & u & \in & (0.0, 1.0] \\
\text{sampling sequence} & s, t & ::= & \epsilon \mid us \\
\text{probability context} & \Gamma & ::= & \cdot \mid \Gamma, x : \tau \\
\text{sampling context} & \Psi & ::= & \emptyset \mid \Psi \cup \{\gamma\}
\end{array}$$

Figure 1: Abstract syntax for λ_γ

$$\frac{x : \tau \in \Gamma}{\Gamma; \Psi \vdash_\gamma x : \tau}\ \mathsf{tp\_var} \qquad \frac{\Gamma, x : \tau; \Psi \vdash_\gamma e : \tau'}{\Gamma; \Psi \vdash_\gamma \lambda x{:}\tau.e : \tau \rightarrow \tau'}\ \mathsf{tp\_lam}$$

$$\frac{\Gamma; \Psi \vdash_\gamma e_1 : \tau' \rightarrow \tau \quad \Gamma; \Psi \vdash_\gamma e_2 : \tau'}{\Gamma; \Psi \vdash_\gamma e_1\ e_2 : \tau}\ \mathsf{tp\_app}$$

$$\frac{\Gamma, x : \tau; \Psi \vdash_\gamma e : \tau'}{\Gamma; \Psi \vdash_\gamma \int x{:}\tau.e : \tau \Rightarrow \tau'}\ \mathsf{tp\_int}$$

$$\frac{\Gamma; \Psi \vdash_\gamma e_1 : \tau' \Rightarrow \tau \quad \Gamma; \Psi \vdash_\gamma e_2 : \tau'}{\Gamma; \Psi \vdash_\gamma e_1 \circ e_2 : \tau}\ \mathsf{tp\_itg}$$

$$\frac{\Gamma; \Psi, \gamma \vdash_\gamma e : \tau}{\Gamma; \Psi \vdash_\gamma \gamma.e : \tau}\ \mathsf{tp\_dist} \qquad \frac{\gamma \in \Psi}{\Gamma; \Psi \vdash_\gamma \gamma : \mathsf{real}}\ \mathsf{tp\_svar}$$

$$\frac{}{\Gamma; \Psi \vdash_\gamma u : \mathsf{real}}\ \mathsf{tp\_real} \qquad \frac{\Gamma, x : \tau; \Psi \vdash_\gamma e : \tau}{\Gamma; \Psi \vdash_\gamma \mathsf{fix}\ x{:}\tau.e : \tau}\ \mathsf{tp\_fix}$$

Figure 2: Typing rules for $\vdash_\gamma$

$$\frac{e_1 \overset{t}{\mapsto} e_1'}{e_1\ e_2 \overset{t}{\mapsto} e_1'\ e_2}\ \mathsf{red\_app} \qquad \frac{}{(\lambda x{:}\tau.e)\ e_2 \overset{\epsilon}{\mapsto} [e_2/x]e}\ \mathsf{red\_lam}$$

$$\frac{e_1 \overset{t}{\mapsto} e_1'}{e_1 \circ e_2 \overset{t}{\mapsto} e_1' \circ e_2}\ \mathsf{red\_itl} \qquad \frac{e_2 \overset{t}{\mapsto} e_2'}{(\int x{:}\tau.e) \circ e_2 \overset{t}{\mapsto} (\int x{:}\tau.e) \circ e_2'}\ \mathsf{red\_itr}$$

$$\frac{}{(\int x{:}\tau.e) \circ v_2 \overset{\epsilon}{\mapsto} [v_2/x]e}\ \mathsf{red\_int}$$

$$\frac{u \sim U(0.0, 1.0]}{\gamma.e \overset{u\epsilon}{\mapsto} [u/\gamma]e}\ \mathsf{red\_dist} \qquad \frac{}{\mathsf{fix}\ x{:}\tau.e \overset{\epsilon}{\mapsto} [\mathsf{fix}\ x{:}\tau.e/x]e}\ \mathsf{red\_fix}$$

Figure 3: Reduction rules for λ_γ

2. CALCULUS λ_γ

2.1 Abstract Syntax

λ_γ arises from the explicitly typed lambda calculus by adding an *integral abstraction* and a sampling construct (Figure 1). An integral abstraction $\int x : \tau.e$ binds a *probability variable* x in its body e as a lambda abstraction does. To apply an integral abstraction to an expression, we use an *integration* $e \circ e$. A sampling construct $\gamma.e$, which binds a *sampling variable* γ in its body e, does not have a corresponding application construct because it substitutes for γ a random number u which is automatically generated by the runtime system. We assume that the name set for probability variables is separate from the name set for sampling variables.

We introduce a new type constructor $\Rightarrow$ for integral abstractions. A sampling construct $\gamma.e$ has the same type as e; hence we do not introduce a new type constructor for sampling constructs. Base types include real for sampling variables and random numbers.

A *sampling sequence* s consists of random numbers in the order that they are generated by the runtime system. ϵ denotes an empty sampling sequence, and $s_1 \cdot s_2$ denotes the concatenation of two sampling sequences. We use t to denote sampling sequences of at most one random number.

2.2 Type System

The type system $\vdash_\gamma$ employs a typing judgment of the form $\Gamma; \Psi \vdash_\gamma e : \tau$, where the typing context consists of a probability context Γ, which is a set of bindings of the form $x : \tau$, and a sampling context Ψ, which is a set of sampling variables (Figure 2). Γ can be augmented by the rules tp_lam and tp_int, and Ψ can be augmented by the rule tp_dist.

2.3 Operational Semantics

We write $[e'/x]e$ for the capture-avoiding substitution of e' for the probability variable x in e. Similarly we write $[u/\gamma]e$ for the capture-free substitution of u for the sampling variable γ in e.

We formulate a structured operational semantics with a reduction judgment of the form $e \overset{t}{\mapsto} e'$ (Figure 3). $e \overset{t}{\mapsto} e'$ means that an expression e reduces to another expression e' by consuming a sampling sequence t. We write $e \overset{s}{\hookrightarrow} v$ if $e \overset{t_1}{\mapsto} e_1 \cdots \overset{t_n}{\mapsto} e_n = v$ and $s = t_1 \cdots t_n$, which means that an expression e evaluates to a value v by consuming a sampling sequence s. $e \overset{s}{\hookrightarrow} v$ also implies that the evaluation of e terminates after consuming s.

The rules red_lam and red_itr indicate that lambda abstractions and integral abstractions assume call-by-name reduction and call-by-value reduction, respectively. By the rule red_dist, the runtime system generates a random number u according to a uniform distribution over the unit interval ($u \sim U(0.0, 1.0]$), and substitute it for the sampling variable γ.

The progress and type preservation properties are stated as follows:

THEOREM 2.1 (PROGRESS). *If $\cdot; \emptyset \vdash_\gamma e : \tau$, then either e is a value or there exists a reduction rule by which e reduces to another expression.*

PROOF. By induction over the structure of $\cdot; \emptyset \vdash_\gamma e : \tau$. □

THEOREM 2.2 (TYPE PRESERVATION). *If $e \overset{t}{\mapsto} e'$ and $\cdot; \emptyset \vdash_\gamma e : \tau$, then $\cdot; \emptyset \vdash_\gamma e' : \tau$.*

We postpone the proof of Theorem 2.2 until we prove the type preservation property of the refinement type system $\vdash_\le$ in Section 3.4.

2.4 Expressiveness

If an expression consumes an empty sampling sequence during its evaluation, it denotes a regular value because the evaluation result is always the same. Otherwise the evaluation entails the use of the rule red_dist at least once and the expression denotes a probability distribution because the evaluation result is determined probabilistically.

Since lambda abstractions and integral abstractions employ different reduction strategies, they exhibit completely different observational behavior even when their bodies are identical. For instance, in the body of a lambda abstraction $\lambda x:\mathsf{real}.x - x$, each occurrence of x denotes an independent probability distribution and hence the resultant probability distribution is not a regular value 0.0 unless its argument denotes a regular value. Therefore it denotes a pure function on probability distributions in the sense that the probability variable x is bound to not a regular value but a probability distribution. In contrast, an integral abstraction $\int x:\mathsf{real}.x - x$ always returns 0.0 regardless of its argument because the two occurrences of x in its body store the same real number. Since the probability variable x is bound to a regular value, it can be thought of as a conditional probability.

The source of probability distributions in λ_γ is the sampling construct $\gamma.e$, which represents a sampling function mapping the unit interval to a probability domain. Note that, in general, the probability domain to which a sampling function maps the unit interval is a set of probability distributions, not regular values, because its body may also denote probability distributions. We can assign a subset of the probability domain a probability proportional to the length of its inverse image under the sampling function. For instance, under a sampling function mapping $(0.0, 0.5]$ to 0 and $(0.5, 1.0]$ to 1, both 0 and 1 are assigned a probability of 0.5 because their inverse images have a length of 0.5.

There are an infinite number of distinct sampling functions which all specify a common probability distribution. A simple method of converting a sampling function to an equivalent one is by rearranging the mapping while preserving the length of the inverse image of any subset of the probability domain. For instance, the following two sampling functions specify the same probability distribution: f_1 mapping $(0.0, 0.5]$ to 0 and $(0.5, 1.0]$ to 1; f_2 mapping $(0.0, 0.5]$ to 1 and $(0.5, 1.0]$ to 0.

There are a variety of methods for expressing probability distributions with sampling functions, most of which are found in the literature of simulation [1]. As a simple example, written in an extension of our calculus, we can encode a uniform distribution over a real interval $(a, b]$ by exploiting the definition of a sampling function:[2]

```
let $uniform = \int a{:}real.\int b{:}real.\gamma.a + \gamma * (b - a)$
```

If a probability distribution is described as a process of generating samples, we can directly encode it by exploiting its definition. The following example encodes a geometric distribution with parameter p in this manner:

```
let $geometric\_definition = \int p:\mathsf{real}.$
  let $bernoulli_p = \gamma.\gamma \leq p$ in
  let rec $geometric_p =$
    if $bernoulli_p$ then 0 else $1 + geometric_p$ in
  $geometric_p$
```

Here $bernoulli_p$ encodes a Bernoulli distribution with parameter p, and a recursive expression $geometric_p$ describes the process of generating a sample from the target geometric distribution. Note that $bernoulli_p$ evaluates to a different sample at each evaluation of $geometric_p$. Alternatively we can encode the same probability distribution by using the inverse of its cumulative distribution function as a sampling function, which is known as the inverse transform method:

```
let $geometric\_inverse = \int p:\mathsf{real}.\gamma.1 + \lfloor \log\ \gamma / \log\ (1.0 - p) \rfloor$
```

The rejection method, which enables us to generate a sample from a probability distribution by repeatedly generating samples from other probability distributions until they satisfy a certain condition, can be easily implemented with recursive expressions. The following example employs this method to encode a Gaussian distribution with mean m and variance s^2:

```
let rec $gaussian\_rejection = \int m{:}real.\int s{:}real.$
  let $bernoulli_{0.5} = \gamma.\gamma \leq 0.5$ in
  let $exponential_{1.0} = \gamma.\gamma - \log\ \gamma$ in
  sample $y_1$ from $exponential_{1.0}$ in
  sample $y_2$ from $exponential_{1.0}$ in
  if $(y_1 - 1.0)^2/2.0 \leq y_2$ then
    if $bernoulli_{0.5}$ then $m + s * y_1$ else $m + s * (-y_1)$
  else
    $gaussian\_rejection \circ m \circ s$
```

Here $exponential_{1.0}$ encodes an exponential distribution with a parameter 1.0 by the inverse transform method.

Sometimes we can employ an ingenious method exploiting unique properties of a target probability distribution. For instance, there is a method for expressing exponential distributions, due to Von Neumann, which requires only the addition and subtraction operations:

```
let $exponential\_Von\_Neumann_{1.0} =$
  let $uniform_{0.0,1.0} = \gamma.\gamma$ in
  let rec $search = \int k:\mathsf{real}.\int u:\mathsf{real}.\int u_1:\mathsf{real}.$
    sample $u'$ from $uniform_{0.0,1.0}$ in
    if $u < u'$ then
      $k + u_1$
    else
      sample $u$ from $uniform_{0.0,1.0}$ in
      if $u \leq u'$ then
        $search \circ k \circ u \circ u_1$
      else
        sample $u$ from $uniform_{0.0,1.0}$ in
        $search \circ (k + 1.0) \circ u \circ u$
    in
  sample $u$ from $uniform_{0.0,1.0}$ in
  $search \circ 0.0 \circ u \circ u$
```

As another example, we can use the central limit theorem to approximate a Gaussian distribution (with a mean 0.0 and a variance 1.0):

```
let $gaussian\_central_{0.0,1.0} =$
  $\gamma_1.\gamma_2.\cdots.\gamma_{12}.(\gamma_1 + \gamma_2 + \cdots + \gamma_{12}) - 6.0$
```

[2]In the examples below, let $x = e$ in e', let rec $x = e$ in e', and sample x from e in e' are syntactic sugar for $(\lambda x:\tau.e')\ e$, $(\lambda x:\tau.e')$ fix $x:\tau.e$, and $(\int x:\tau.e') \circ e$, respectively, where τ is an appropriate type for e.

By virtue of lack of any semantic restriction on the body of a sampling construct, we can also express unusual probability distributions over unusual domains. For instance, we can express a combination of a Dirac distribution and a uniform distribution over the same domain as follows:

$$\mathsf{let}\ \mathit{dirac\_uniform} = \gamma.\mathsf{if}\ \gamma < 0.5\ \mathsf{then}\ 0.0\ \mathsf{else}\ \gamma$$

As another example, we can implement the addition of two probability distributions over angular space in a straightforward way:

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

Note that if we chose probability density functions, for instance, as the mathematical basis for λ_γ, neither $\mathit{dirac\_uniform}$ nor $\mathit{add\_angle}$ could be easily implemented: in the case of $\mathit{dirac\_uniform}$, there is no way to assign a nonzero probability to a single real number, and in the case of $\mathit{add\_angle}$, we have to take into account the fact that an angle θ is identified with $\theta + 2\pi$.

All these examples are evidence of the versatility of λ_γ: *the more we know about a probability distribution, the better we can encode it.*

2.5 Implementation Issues

In implementing the operational semantics, we can simulate lambda abstractions and applications with integral abstractions and integrations (as we simulate lazy evaluation with eager evaluation) by a mapping $|\cdot|$

$$\begin{array}{rcl} |\tau_1 \rightarrow \tau_2| & = & (\mathsf{1} \Rightarrow |\tau_1|) \Rightarrow |\tau_2| \\ |\lambda x : \tau . e| & = & \int x : \mathsf{1} \Rightarrow |\tau| . |[(x \circ \langle\rangle)/x]e| \\ |e_1\ e_2| & = & |e_1| \circ \int \_ : \mathsf{1} . |e_2| \end{array}$$

where $\mathsf{1}$ is a base type, $\langle\rangle$ is a value of type $\mathsf{1}$, and $\_$ is a dummy probability variable. Then it may appear that lambda abstractions and applications of λ_γ are redundant. However the type $(\mathsf{1} \Rightarrow |\tau_1|) \Rightarrow |\tau_2|$ does not deliver the intended meaning of the type $\tau_1 \rightarrow \tau_2$, namely, pure functions on probability distributions; instead it represents conditional probabilities whose intuitive meaning is not easy to describe. Moreover a lambda abstraction may impose on its argument or body a restriction which an integral abstraction does not. For instance, when we extend the language with effects, we may syntactically require that an argument to a lambda abstraction be free of effects while an integral abstraction accepts any expression as its argument. Then such a restriction cannot be enforced if we employ the mapping $|\cdot|$. Therefore we choose to leave lambda abstractions and applications as a core part of λ_γ.

Since a sampling variable γ ranges over the unit interval, a full implementation of the operational semantics requires real arithmetic at an arbitrary precision. Although there are well-developed theories for exact real arithmetic (see [4] for an example), we do not find it necessary to incorporate such an expensive functionality into the implementation of the operational semantics sacrificing the efficiency. The reason is that often probabilistic computations modeled by probabilistic programs themselves are not specified accurately. For instance, in the robot localization problem, which is to estimate the pose of a robot from sensor readings, there is no point in employing exact real arithmetic because all sensor readings are floating point numbers.

$$\frac{}{b :: b}\ \mathsf{ref\_base} \qquad \frac{\sigma :: \tau}{\Box\sigma :: \tau}\ \mathsf{ref\_box} \qquad \frac{\sigma_1 :: \tau \quad \sigma_2 :: \tau}{\sigma_1 \wedge \sigma_2 :: \tau}\ \mathsf{ref\_inter}$$

$$\frac{\sigma_1 :: \tau_1 \quad \sigma_2 :: \tau_2}{\sigma_1 \rightarrow \sigma_2 :: \tau_1 \rightarrow \tau_2}\ \mathsf{ref\_lam} \qquad \frac{\sigma_1 :: \tau_1 \quad \sigma_2 :: \tau_2}{\sigma_1 \Rightarrow \sigma_2 :: \tau_1 \Rightarrow \tau_2}\ \mathsf{ref\_int}$$

Figure 4: Refinement rules

Another important consideration is that the type system $\vdash_\gamma$ lacks the capability to check whether an expression denotes a regular value or not; it infers only the probability domain of a given expression. Such a capability is desirable to achieve an efficient implementation of the operational semantics: we can evaluate an expression without incurring an overhead of probabilistic computation if it is known to denote a regular value. Therefore we enhance the type system $\vdash_\gamma$ in order to statically distinguish expressions denoting regular values from expressions denoting probability distributions, which is the topic of the next section.

3. REFINEMENT TYPE SYSTEM

In this section, we enrich the previous type system to obtain a refinement type system $\vdash_\leq$ in which regular values are interpreted as belonging to subtypes of probability distributions. First we formulate a subtyping system in the customary way as a binary relation between types. Then we revise it into a structural subtyping system. The two subtyping systems are shown to be equivalent, but the second system is easier to reason about. Finally we prove the type preservation property of the new type system and its soundness.

3.1 Syntax for Refinement Types

A refinement type σ is constructed from a type τ with a modal connective $\Box$ and an intersection connective $\wedge$:

$$\text{refinement type} \quad \sigma, \rho \quad ::= \quad b \mid \sigma \rightarrow \sigma \mid \sigma \Rightarrow \sigma \mid \Box\sigma \mid \sigma \wedge \sigma$$

In order to state that a refinement type σ refines a type τ, we introduce a refinement judgment of the form $\sigma :: \tau$ (Figure 4). $\sigma :: \tau$ means that we can recover τ from σ by removing $\Box$ and collapsing conjuncts of $\wedge$. We say that a refinement type σ is well-formed if there exists a type τ such that $\sigma :: \tau$.

3.2 Subtyping

A modal type $\Box\sigma$ denotes the set of regular values among those probability distributions denoted by σ. Since a regular value is a special form of probability distribution, we obtain a subtyping principle $\Box\sigma \leq \sigma$. An intersection type $\sigma_1 \wedge \sigma_2$ denotes the intersection of the sets of probability distributions denoted by σ_1 and σ_2.

The subtyping relation $\leq$ is defined by a subtyping judgment of the form $\sigma \leq \sigma'$, which means that if an expression has type σ, it also has type σ' (Figure 5). By the rules $\mathsf{sub\_lam}$ and $\mathsf{sub\_int}$, the type constructors $\rightarrow$ and $\Rightarrow$ are contravariant in the argument and covariant in the result. The rule $\mathsf{sub\_dbox}$ implies that we can abbreviate $\Box\Box\sigma$ into $\Box\sigma$ because $\Box\Box\sigma$ is also a subtype of $\Box\sigma$ by the rule $\mathsf{sub\_def}$. The distributivity rule $\mathsf{sub\_dist}$ implies that we can expand $\Box(\sigma \wedge \sigma')$ into $\Box\sigma \wedge \Box\sigma'$ because $\Box(\sigma \wedge \sigma')$ can also

$$\frac{}{\sigma \leq \sigma}\ \textsf{sub\_id} \qquad \frac{\sigma_1 \leq \sigma_2 \quad \sigma_2 \leq \sigma_3}{\sigma_1 \leq \sigma_3}\ \textsf{sub\_trans}$$

$$\frac{\sigma_1' \leq \sigma_1 \quad \sigma_2 \leq \sigma_2'}{\sigma_1 \rightarrow \sigma_2 \leq \sigma_1' \rightarrow \sigma_2'}\ \textsf{sub\_lam} \qquad \frac{\sigma_1' \leq \sigma_1 \quad \sigma_2 \leq \sigma_2'}{\sigma_1 \Rightarrow \sigma_2 \leq \sigma_1' \Rightarrow \sigma_2'}\ \textsf{sub\_int}$$

$$\frac{}{\Box\sigma \leq \sigma}\ \textsf{sub\_def} \qquad \frac{}{\Box\sigma \leq \Box\Box\sigma}\ \textsf{sub\_dbox} \qquad \frac{\sigma \leq \sigma'}{\Box\sigma \leq \Box\sigma'}\ \textsf{sub\_box}$$

$$\frac{\sigma_1 \wedge \sigma_2 :: \tau}{\sigma_1 \wedge \sigma_2 \leq \sigma_1}\ \textsf{sub\_inter\_l} \qquad \frac{\sigma_1 \wedge \sigma_2 :: \tau}{\sigma_1 \wedge \sigma_2 \leq \sigma_2}\ \textsf{sub\_inter\_r}$$

$$\frac{\sigma \leq \sigma_1 \quad \sigma \leq \sigma_2}{\sigma \leq \sigma_1 \wedge \sigma_2}\ \textsf{sub\_inter} \qquad \frac{}{\Box\sigma \wedge \Box\sigma' \leq \Box(\sigma \wedge \sigma')}\ \textsf{sub\_dist}$$

Figure 5: Subtyping rules

be shown to be a subtype of $\Box\sigma \wedge \Box\sigma'$. The premise of the rules sub_inter_l and sub_inter_r is necessary to enforce that if $\sigma \leq \sigma'$, then both σ and σ' are well-formed with respect to a common type.

Although we include a distributivity rule for the connective $\Box$, we omit distributivity rules for the type constructors $\rightarrow$ and $\Rightarrow$, the second of which is unsound in the presence of effects [2]:

$$\frac{}{(\sigma \rightarrow \sigma_1) \wedge (\sigma \rightarrow \sigma_2) \leq \sigma \rightarrow (\sigma_1 \wedge \sigma_2)}\ \textsf{sub\_lam\_dist}$$

$$\frac{}{(\sigma \Rightarrow \sigma_1) \wedge (\sigma \Rightarrow \sigma_2) \leq \sigma \Rightarrow (\sigma_1 \wedge \sigma_2)}\ \textsf{sub\_int\_dist}$$

We define an equivalence relation $\equiv$ such that $\sigma \equiv \sigma'$ if and only if $\sigma \leq \sigma'$ and $\sigma' \leq \sigma$. The following equations allow us to simplify refinement types (for instance, by removing parentheses and $\Box$ as necessary):

$$\begin{array}{llcl} (a) & \sigma \wedge \sigma & \equiv & \sigma \\ (b) & \sigma \wedge \sigma' & \equiv & \sigma' \wedge \sigma \\ (c) & \sigma \wedge (\sigma' \wedge \sigma'') & \equiv & (\sigma \wedge \sigma') \wedge \sigma'' \\ (d) & \Box\Box\sigma & \equiv & \Box\sigma \\ (e) & \Box(\sigma \wedge \sigma') & \equiv & \Box\sigma \wedge \Box\sigma' \end{array}$$

Note that there are only a finite number of refinement types which refine a common type and are distinct modulo $\equiv$. For instance, the type real $\rightarrow$ real has six refinement types which are distinct modulo $\equiv$: $\Box(\textsf{real} \rightarrow \textsf{real})$, $\Box(\textsf{real} \rightarrow \textsf{real}) \wedge \Box(\Box\textsf{real} \rightarrow \Box\textsf{real})$, $\Box(\textsf{real} \rightarrow \Box\textsf{real})$, $\textsf{real} \rightarrow \textsf{real}$, $(\textsf{real} \rightarrow \textsf{real}) \wedge (\Box\textsf{real} \rightarrow \Box\textsf{real})$, and $\textsf{real} \rightarrow \Box\textsf{real}$. Therefore it is straightforward to show that provability of $\sigma \leq \sigma'$ is decidable.

Although we can decide $\sigma \leq \sigma'$ for any refinement types σ and σ', it is not easy to reason about the subtyping system, especially due to the transitivity rule sub_trans and the distributivity rule sub_dist. For instance, it is not obvious how to prove the following simple propositions, which are used in Section 3.4:

PROPOSITION 3.1. *If $\Box(\sigma_1 \rightarrow \sigma_2) \leq \Box(\sigma_1' \rightarrow \sigma_2')$ or $\Box(\sigma_1 \Rightarrow \sigma_2) \leq \Box(\sigma_1' \Rightarrow \sigma_2')$, then $\sigma_1' \leq \sigma_1$ and $\sigma_2 \leq \sigma_2'$.*

PROPOSITION 3.2. *If $\sigma \leq \Box\sigma'$, then 1) $\sigma = \Box\rho$; or 2) $\sigma = \sigma_1 \wedge \sigma_2$ and either $\sigma_1 \leq \Box\rho$ or $\sigma_2 \leq \Box\rho$, for a certain refinement type ρ.*

In the next subsection, we formulate a structural subtyping system which is equivalent to the above subtyping system yet easier to reason about.

3.3 Structural Subtyping

The previous subtyping system is not orthogonal in the sense that the connectives $\Box$ and $\wedge$ are dependent on each other via the distributivity rule sub_dist. Lack of orthogonality makes it difficult to prove various properties of the subtyping relation $\leq$ and eventually leads to inability to prove the type preservation property and the soundness of the type system $\vdash_{\leq}$ to be developed later. Still we cannot simply drop the distributivity rule at the expense of expressive power because the equation (e) seems to be indispensable.

To remedy this dilemma, we propose a structural subtyping relation $\preceq$. The structural subtyping judgment has the form $\Lambda; \Sigma \preceq \sigma$ where Λ and Σ are separate subtyping contexts (Figure 6):

$$\begin{array}{lrcl} \text{regular value context} & \Lambda & ::= & \cdot \mid \Lambda, \sigma \\ \text{probability distribution context} & \Sigma & ::= & \cdot \mid \Sigma, \sigma \end{array}$$

We assume that all refinement types present in $\Lambda; \Sigma \preceq \sigma$ are well-formed with respect to a common type.

The intuition here is that $\sigma_1, \cdots, \sigma_m; \sigma_1', \cdots, \sigma_n' \preceq \rho$ means $\Box\sigma_1 \wedge \cdots \wedge \Box\sigma_m \wedge \sigma_1' \wedge \cdots \wedge \sigma_n' \leq \rho$: if an expression has types $\Box\sigma_1, \cdots, \Box\sigma_m, \sigma_1, \cdots$, and σ_n, then it also has type ρ. In other words, we implicitly assume the use of $\Box$ for each refinement type in Λ and the use of $\wedge$ among all refinement types in Λ and Σ. For instance, the rules log_box and log_inter conform to this intuition.

Since every rule involves only one type constructor or connective, the structural subtyping system is orthogonal. The interaction between $\Box$ and $\wedge$ in the system, which is expressed explicitly by the distributivity rule sub_dist in the previous subtyping system, is now expressed by the implicit use of $\Box$ and $\wedge$. For instance, the rule log_inter_v implies $\Box(\sigma_1 \wedge \sigma_2) \leq \Box\sigma_1 \wedge \Box\sigma_2$ in the previous subtyping system.

When we apply the rule log_lam_v or log_lam to derive a judgment $\Lambda; \Sigma \preceq \rho \rightarrow \rho'$, only one type in Λ and Σ is compared with $\rho \rightarrow \rho'$. Therefore, for instance, even if an expression has types $\sigma \rightarrow \sigma_1'$ and $\sigma \rightarrow \sigma_2'$, we must select either type when we apply the rule log_lam in order to show that it also has type $\rho \rightarrow \rho'$; we cannot use both types together. In the case of $\rho \rightarrow \rho' = \sigma \rightarrow \sigma_1' \wedge \sigma_2'$, this means that the structural subtyping system omits a distributivity rule for the type constructor $\rightarrow$ similar to the rule sub_lam_dist. If we are to support a general distributivity rule for $\rightarrow$, we must include new rules capable of inspecting multiple types simultaneously such as

$$\frac{\cdot; \rho \preceq \sigma_1 \quad \cdots \quad \cdot; \rho \preceq \sigma_n \quad \cdot; \sigma_1' \preceq \rho' \quad \cdots \quad \cdot; \sigma_n' \preceq \rho'}{\Lambda; \Sigma, \sigma_1 \rightarrow \sigma_1', \cdots, \sigma_n \rightarrow \sigma_n' \preceq \rho \rightarrow \rho'}\ \textsf{log\_lam\_dist}$$

A similar argument can be made for a general distributivity rule for the type constructor $\Rightarrow$.

The structural subtyping system resembles the system for the intuitionistic modal logic S4 of Pfenning and Davies [14], which is presented in the style of natural deduction. Their system separates valid hypotheses from true hypotheses, and a judgment has the form $\Delta; \Gamma \vdash J$ where the antecedent consists of Δ for valid hypotheses and Γ for true hypotheses. A true hypothesis $\Box A$ can be used as a valid hypothesis A through the elimination rule for $\Box$, which is analogous to the

$$\frac{}{\Lambda, b; \Sigma \preceq b}\ \textsf{log\_base\_v} \qquad \frac{}{\Lambda; \Sigma, b \preceq b}\ \textsf{log\_base} \qquad \frac{\cdot; \sigma_1' \preceq \sigma_1 \quad \cdot; \sigma_2 \preceq \sigma_2'}{\Lambda, \sigma_1 \rightarrow \sigma_2; \Sigma \preceq \sigma_1' \rightarrow \sigma_2'}\ \textsf{log\_lam\_v} \qquad \frac{\cdot; \sigma_1' \preceq \sigma_1 \quad \cdot; \sigma_2 \preceq \sigma_2'}{\Lambda; \Sigma, \sigma_1 \rightarrow \sigma_2 \preceq \sigma_1' \rightarrow \sigma_2'}\ \textsf{log\_lam}$$

$$\frac{\cdot; \sigma_1' \preceq \sigma_1 \quad \cdot; \sigma_2 \preceq \sigma_2'}{\Lambda, \sigma_1 \Rightarrow \sigma_2; \Sigma \preceq \sigma_1' \Rightarrow \sigma_2'}\ \textsf{log\_int\_v} \qquad \frac{\cdot; \sigma_1' \preceq \sigma_1 \quad \cdot; \sigma_2 \preceq \sigma_2'}{\Lambda; \Sigma, \sigma_1 \Rightarrow \sigma_2 \preceq \sigma_1' \Rightarrow \sigma_2'}\ \textsf{log\_int} \qquad \frac{\Lambda, \sigma; \Sigma \preceq \sigma'}{\Lambda, \Box\sigma; \Sigma \preceq \sigma'}\ \textsf{log\_box\_v} \qquad \frac{\Lambda, \sigma; \Sigma \preceq \sigma'}{\Lambda; \Sigma, \Box\sigma \preceq \sigma'}\ \textsf{log\_box}$$

$$\frac{\Lambda; \cdot \preceq \sigma}{\Lambda; \Sigma \preceq \Box\sigma}\ \textsf{log\_box\_r} \qquad \frac{\Lambda, \sigma_1, \sigma_2; \Sigma \preceq \sigma}{\Lambda, \sigma_1 \wedge \sigma_2; \Sigma \preceq \sigma}\ \textsf{log\_inter\_v} \qquad \frac{\Lambda; \Sigma, \sigma_1, \sigma_2 \preceq \sigma}{\Lambda; \Sigma, \sigma_1 \wedge \sigma_2 \preceq \sigma}\ \textsf{log\_inter} \qquad \frac{\Lambda; \Sigma \preceq \sigma_1 \quad \Lambda; \Sigma \preceq \sigma_2}{\Lambda; \Sigma \preceq \sigma_1 \wedge \sigma_2}\ \textsf{log\_inter\_r}$$

Figure 6: Structural subtyping rules

rule log_box in our system. As Davies and Pfenning [3] associate $\Box A$ with closed code producing a value of type A via the Curry-Howard isomorphism, we can interpret a refinement type $\Box\sigma$ as the set of *closed* expressions of refinement type σ *in the sense that their evaluation does not depend on sampling sequences*, which coincides with the intuition behind the modal connective $\Box$. Note, however, that this interpretation is not based upon the Curry-Howard isomorphism because expressions of λ_γ are not proof terms for the structural subtyping system.

The rules for the connectives $\Box$ and $\wedge$ are presented in the style of Gentzen's sequent calculus: for each connective, there are two rules for introducing the connective on the left side and one rule for introducing the connective on the right side of a structural subtyping judgment. The rules for base types correspond to the rule for initial sequents in Gentzen's sequent calculi. The rules for the type constructors $\rightarrow$ and $\Rightarrow$ can also be thought of as corresponding to the rule for initial sequents because their premise contains only refinement types enclosed by a type constructor in their conclusion, which are secure from the rules for the connectives. In other words, they serve as an interface between two independent derivations for $\Box$ and $\wedge$.

The structural subtyping system does not have any transitivity rule. However we can prove *admissibility of the cut rules*, which states that the following *cut rules* are redundant in our system. Note that the cut rules do not involve any connective.

$$\frac{\Lambda; \cdot \preceq \sigma \quad \Lambda, \sigma; \Sigma \preceq \sigma'}{\Lambda; \Sigma \preceq \sigma'}\ \textsf{log\_cut\_v} \qquad \frac{\Lambda; \Sigma \preceq \sigma \quad \Lambda; \Sigma, \sigma \preceq \sigma'}{\Lambda; \Sigma \preceq \sigma'}\ \textsf{log\_cut}$$

The proof is completely analogous to the proof in the Gentzen's sequent calculus for intuitionistic propositional logic. Unlike intuitionistic propositional logic, however, our system does not have contraction rules, and thus we need to prove the contraction property. We first prove admissibility of identity rules and the weakening property.

Proposition 3.3. *For any Λ, Σ, and σ, it holds $\Lambda, \sigma; \Sigma \preceq \sigma$ and $\Lambda; \Sigma, \sigma \preceq \sigma$.*

Lemma 3.4 (Weakening). *If $\Lambda; \Sigma \preceq \sigma$, then $\Lambda, \sigma'; \Sigma \preceq \sigma$ and $\Lambda; \Sigma, \sigma' \preceq \sigma$.*

Proof. By induction over the structure of $\Lambda; \Sigma \preceq \sigma$. $\Box$

In the above lemma, both derivations in the conclusion employ the same set of structural subtyping rules and hence have the same size as the derivation in the assumption.

Lemma 3.5 (Contraction). *If $\Lambda; \Sigma, \sigma, \sigma \preceq \sigma'$, then $\Lambda; \Sigma, \sigma \preceq \sigma'$, and if $\Lambda, \sigma, \sigma; \Sigma \preceq \sigma'$, then $\Lambda, \sigma; \Sigma \preceq \sigma'$.*

Proof. By nested induction on the structure of σ and the structure of $\Lambda; \Sigma, \sigma, \sigma \preceq \sigma'$ or $\Lambda, \sigma, \sigma; \Sigma \preceq \sigma'$. $\Box$

In the above lemma, the derivation in the conclusion has the same or smaller size than the derivation in the assumption. The proof of admissibility of the cut rules relies on the size of derivations because we do not have proof terms for the structural subtyping system.

Theorem 3.6 (Admissibility of the Cut Rules). *If $\Lambda; \Sigma \preceq \sigma$ and $\Lambda; \Sigma, \sigma \preceq \sigma'$, then $\Lambda; \Sigma \preceq \sigma'$, and if $\Lambda; \cdot \preceq \sigma$ and $\Lambda, \sigma; \Sigma \preceq \sigma'$, then $\Lambda; \Sigma \preceq \sigma'$.*

Proof. By three nested induction on the structure of σ and the size of $\Lambda; \Sigma \preceq \sigma$ and $\Lambda; \Sigma, \sigma \preceq \sigma'$, or $\Lambda; \cdot \preceq \sigma$ and $\Lambda, \sigma; \Sigma \preceq \sigma'$. $\Box$

Provability in the structural subtyping system is decidable because it satisfies the subformula property: in any cut-free derivation of a judgment, only subformulae of the formulae occurring in the judgment can occur. We can also obtain an algorithmic structural subtyping system as follows. In order to derive a judgment $\Lambda; \Sigma \preceq \sigma$, we first apply backwards the rules for $\Box$ and $\wedge$ until no more connectives can be eliminated; if any connective is left, it must be enclosed within a type constructor. We apply the rule log_box_r only when no other rules can be applied. Then we apply either the rules for bases types to close the derivation, or the rules for $\rightarrow$ and $\Rightarrow$ to initiate the derivation of a smaller judgment.

Now we prove that the structural subtyping system is equivalent to the previous subtyping system:

Theorem 3.7. *$\cdot; \sigma \preceq \sigma'$ if and only if $\sigma \leq \sigma'$.*

The proof of Theorem 3.7 employs two propositions. Proposition 3.8 states that the previous subtyping system is sound with respect to the structural subtyping system. In other words, if we can derive $\sigma \leq \sigma'$ under the previous subtyping system, then we can also derive $\cdot; \sigma \preceq \sigma'$ under the structural subtyping system. Proposition 3.9 states that the previous subtyping system is complete with respect to the structural subtyping system.

Proposition 3.8. *All the subtyping rules* sub_id *through* sub_dist *in Figure 5 are sound if we interpret $\sigma \leq \sigma'$ as $\cdot; \sigma \preceq \sigma'$.*

Proposition 3.9. *If $\cdot; \sigma \preceq \sigma'$, then $\sigma \leq \sigma'$.*

Proposition 3.9 is obtained as a corollary to Lemma 3.10, which states precisely the intuition behind the structural subtyping relation $\preceq$.

LEMMA 3.10. *If $\sigma_1, \cdots, \sigma_m; \sigma_1', \cdots, \sigma_n' \preceq \rho$, then $\Box\sigma_1 \wedge \cdots \wedge \Box\sigma_m \wedge \sigma_1' \wedge \cdots \wedge \sigma_n' \leq \rho$.*

PROOF. By induction over the structure of the derivation $\sigma_1, \cdots, \sigma_m; \sigma_1', \cdots, \sigma_n' \preceq \rho$. $\Box$

The equivalence between the two subtyping systems with respect to $\leq$ is significant independently of the main topic of this paper. The previous subtyping system can be thought of as an an axiomatic characterization of the entailment for the intuitionistic modal logic S4 with an intersection connective. For instance, the rules sub_def, sub_dbox, and sub_box correspond to the following axioms:

$$\begin{array}{ll} \vdash & \Box A \supset A \\ \vdash & \Box A \supset \Box\Box A \\ \vdash & \Box(A \supset B) \supset (\Box A \supset \Box B) \end{array}$$

The premise of the rule sub_box can be interpreted as $\Box(A \supset B)$ because it has no hypotheses. Then, by the equivalence between the two subtyping systems, we obtain another formulation of the same logic with much better properties. For instance, it can be easily extended with new connectives because of its orthogonality. In contrast, in order to add a new connective to the previous subtyping system, we must examine closely its interaction with all existing connectives, which often turns out be a difficult task.

From now on, we use $\sigma \leq \sigma'$ and $\cdot; \sigma \preceq \sigma'$ interchangeably. We close this subsection by proving Proposition 3.1 and Proposition 3.2 to demonstrate that it is easy to prove various properties of the subtyping relation $\leq$ under the structural subtyping system.

PROOF OF PROPOSITION 3.1. The only way to derive $\Box(\sigma_1 \to \sigma_2) \leq \Box(\sigma_1' \to \sigma_2')$ without employing the cut rules is

$$\dfrac{\dfrac{\dfrac{\cdot; \sigma_1' \preceq \sigma_1 \quad \cdot; \sigma_2 \preceq \sigma_2'}{\sigma_1 \to \sigma_2; \cdot \preceq (\sigma_1' \to \sigma_2')}\ \textsf{log\_lam\_v}}{\sigma_1 \to \sigma_2; \cdot \preceq \Box(\sigma_1' \to \sigma_2')}\ \textsf{log\_box\_r}}{\cdot; \Box(\sigma_1 \to \sigma_2) \preceq \Box(\sigma_1' \to \sigma_2')}\ \textsf{log\_box}$$

Therefore $\sigma_1' \leq \sigma_1$ and $\sigma_2 \leq \sigma_2'$ follow from $\cdot; \sigma_1' \preceq \sigma_1$ and $\cdot; \sigma_2 \preceq \sigma_2'$, respectively. The case of $\Box(\sigma_1 \Rightarrow \sigma_2) \leq \Box(\sigma_1' \Rightarrow \sigma_2')$ is similar. $\Box$

PROOF OF PROPOSITION 3.2. If σ is b, $\sigma_1 \to \sigma_2$, or $\sigma_1 \Rightarrow \sigma_2$, then there exists no derivation $\cdot; \sigma \preceq \Box\sigma'$. If $\sigma = \sigma_1 \wedge \sigma_2$, then $\sigma_1 \wedge \sigma_2 \leq \Box\sigma'$ is expanded to

$$\dfrac{\dfrac{\dfrac{\Lambda_1, \Lambda_2; \cdot \preceq \sigma'}{\Lambda_1, \Lambda_2; \Sigma_1, \Sigma_2 \preceq \Box\sigma'}\ \textsf{log\_box\_r}}{\cdot; \sigma_1, \sigma_2 \preceq \Box\sigma'}}{\cdot; \sigma_1 \wedge \sigma_2 \preceq \Box\sigma'}\ \textsf{log\_inter}$$

where σ_1 and σ_2 are expanded to Λ_1, Σ_1 and Λ_2, Σ_2, respectively. Since Λ_1, Λ_2 is not empty, there exists an application of log_box in the derivation. Therefore the fragment of the derivation from the application of log_inter to the first application of log_box is written as

$$\dfrac{\dfrac{\rho; \Sigma \preceq \Box\sigma'}{\cdot; \sigma_{11}, \cdots, \sigma_{1m}, \sigma_{21}, \cdots, \sigma_{2n} \preceq \Box\sigma'}\ \textsf{log\_box}}{\cdot; \sigma_1, \sigma_2 \preceq \Box\sigma'}$$

where σ_1 and σ_2 are expanded to $\sigma_{11}, \cdots, \sigma_{1m}$ and $\sigma_{21}, \cdots, \sigma_{2n}$, respectively, by log_inter only. Suppose $\sigma_{11} = \Box\rho$ and $\Sigma = \sigma_{12}, \cdots, \sigma_{1m}, \sigma_{21}, \cdots, \sigma_{2n}$. Then $\sigma_1 \leq \Box\rho$ by

$$\dfrac{\dfrac{\rho; \sigma_{12}, \cdots, \sigma_{1m} \preceq \rho}{\cdot; \sigma_{11}, \cdots, \sigma_{1m} \preceq \Box\rho}\ \textsf{log\_box}}{\cdot; \sigma_1 \preceq \Box\rho}$$

In a similar way, we can show that $\sigma_1 \leq \Box\rho$ or $\sigma_2 \leq \Box\rho$ always holds. $\Box$

3.4 Refinement Type System

Now we formulate a refinement type system $\vdash_{\leq}$. A refinement typing judgment has the form $\Delta; \Psi \vdash_{\leq} e : \sigma$ where a refinement probability context Δ is a set of bindings of the form $x : \sigma$ (Figure 7):

$$\text{refinement probability context} \quad \Delta \quad ::= \quad \cdot \mid \Delta, x : \sigma$$

$up(\sigma)$ removes leading $\Box$'s from σ if any:

$$\begin{array}{rcl} up(b) & = & b \\ up(\sigma_1 \to \sigma_2) & = & \sigma_1 \to \sigma_2 \\ up(\sigma_1 \Rightarrow \sigma_2) & = & \sigma_1 \Rightarrow \sigma_2 \\ up(\Box\sigma) & = & up(\sigma) \\ up(\sigma_1 \wedge \sigma_2) & = & up(\sigma_1) \wedge up(\sigma_2) \end{array}$$

The implication is that if σ denotes a set D of regular values (for instance, $\sigma = \Box\textsf{real}$), $up(\sigma)$ augments the set with those probability distributions whose probability domain is D. In the case that σ denotes a set of probability distributions including regular values (for instance, $\sigma = \textsf{real}$), $up(\sigma)$ is equal to σ. Since the set denoted by σ is contained by the set denoted by $up(\sigma)$, σ must be a subtype of $up(\sigma)$:

PROPOSITION 3.11. *If σ is well-formed, then $\sigma \leq up(\sigma)$.*

The refinement typing rules are all in accordance with the intended meaning of the function $up(\cdot)$ as well as the connectives $\Box$ and $\wedge$. For instance, in the rules ts_app and ts_itg, the type in the conclusion is not σ' but $up(\sigma')$ because the expression e_1, whose type is not proven to be a modal type of the form $\Box\cdot$, may consume random numbers during its evaluation. The rule ts_dist states that a sampling construct $\gamma.e$ cannot denote a regular value. The refinement typing rules also take into account the operational semantics of λ_γ. For instance, in contrast to the rule tp_int in the type system $\vdash_\gamma$, the rule ts_int now exploits the fact that in the body of an integral abstraction $\int x : \tau.e$, the probability variable x is bound to a regular value. In the rule ts_svar, the type in the conclusion is not $\Box\textsf{real}$ but $\textsf{real}$ because the result of evaluating the sampling construct γ is probabilistic.

As mentioned in Section 3.2, there are only a finite number of refinement types which refine a common type and are distinct modulo $\equiv$. It implies that we only need to try a finite number of derivations to see if $\Delta; \Psi \vdash_{\leq} e : \sigma$ holds. Therefore the type system $\vdash_{\leq}$ is decidable.

Type Preservation

Since an expression may consume random numbers by the rule red_dist during its evaluation, it gets closer to a value as its evaluation proceeds, and consequently its type also becomes more accurate with respect to the subtyping relation $\leq$. Intuitively, if $e \overset{t}{\mapsto} e'$, then $t = \epsilon$ implies that e' can be assigned the same type as e whereas $t = u\epsilon$ implies that the type σ' of e' must be a subtype of the type σ of e. In the second case, however, we cannot decide whether or not σ' is a strict subtype of σ, that is, $\sigma' \leq \sigma$ but $\sigma \not\leq \sigma'$. Therefore

$$\frac{\Delta;\Psi \vdash_{\leq} e:\sigma \quad \sigma \leq \sigma'}{\Delta;\Psi \vdash_{\leq} e:\sigma'}\ \textsf{ts\_trans} \qquad \frac{\Delta;\Psi \vdash_{\leq} e:\sigma_1 \quad \Delta;\Psi \vdash_{\leq} e:\sigma_2}{\Delta;\Psi \vdash_{\leq} e:\sigma_1 \wedge \sigma_2}\ \textsf{ts\_inter} \qquad \frac{x:\sigma \in \Delta}{\Delta;\Psi \vdash_{\leq} x:\sigma}\ \textsf{ts\_var}$$

$$\frac{\Delta, x:\sigma;\Psi \vdash_{\leq} e:\sigma' \quad \sigma :: \tau}{\Delta;\Psi \vdash_{\leq} \lambda x{:}\tau.e : \Box(\sigma \rightarrow \sigma')}\ \textsf{ts\_lam} \quad \frac{\Delta;\Psi \vdash_{\leq} e_1:\sigma \rightarrow \sigma' \quad \Delta;\Psi \vdash_{\leq} e_2:\sigma}{\Delta;\Psi \vdash_{\leq} e_1\ e_2 : \mathit{up}(\sigma')}\ \textsf{ts\_app} \quad \frac{\Delta;\Psi \vdash_{\leq} e_1:\Box(\sigma \rightarrow \sigma') \quad \Delta;\Psi \vdash_{\leq} e_2:\sigma}{\Delta;\Psi \vdash_{\leq} e_1\ e_2 : \sigma'}\ \textsf{ts\_app\_b}$$

$$\frac{\Delta, x:\Box\sigma;\Psi \vdash_{\leq} e:\sigma' \quad \sigma :: \tau}{\Delta;\Psi \vdash_{\leq} \int x{:}\tau.e : \Box(\sigma \Rightarrow \sigma')}\ \textsf{ts\_int} \quad \frac{\Delta;\Psi \vdash_{\leq} e_1:\sigma \Rightarrow \sigma' \quad \Delta;\Psi \vdash_{\leq} e_2:\sigma}{\Delta;\Psi \vdash_{\leq} e_1 \circ e_2 : \mathit{up}(\sigma')}\ \textsf{ts\_itg} \quad \frac{\Delta;\Psi \vdash_{\leq} e_1:\Box(\sigma \Rightarrow \sigma') \quad \Delta;\Psi \vdash_{\leq} e_2:\Box\sigma}{\Delta;\Psi \vdash_{\leq} e_1 \circ e_2 : \sigma'}\ \textsf{ts\_itg\_b}$$

$$\frac{\Delta;\Psi \cup \{\gamma\} \vdash_{\leq} e:\sigma}{\Delta;\Psi \vdash_{\leq} \gamma.e : \mathit{up}(\sigma)}\ \textsf{ts\_dist} \qquad \frac{\gamma \in \Psi}{\Delta;\Psi \vdash_{\leq} \gamma : \textsf{real}}\ \textsf{ts\_svar} \qquad \frac{}{\Delta;\Psi \vdash_{\leq} u : \Box\textsf{real}}\ \textsf{ts\_real} \qquad \frac{\Delta, x:\sigma;\Psi \vdash_{\leq} e:\sigma \quad \sigma :: \tau}{\Delta;\Psi \vdash_{\leq} \textsf{fix}\ x{:}\tau.e : \sigma}\ \textsf{ts\_fix}$$

Figure 7: Refinement typing rules for $\vdash_{\leq}$

we choose a conservative definition of the type preservation property for $\vdash_{\leq}$, which states that e' can be assigned at least the same type as e:

THEOREM 3.12 (TYPE PRESERVATION). *If $e \overset{t}{\mapsto}_{\leq} e'$, then $\cdot;\emptyset \vdash_{\leq} e:\sigma$ implies $\cdot;\emptyset \vdash_{\leq} e':\sigma$.*

The proof of the type preservation property requires two propositions regarding substitution and two lemmas. Lemma 3.15 and Lemma 3.17 exploit Proposition 3.1 in their proof.

PROPOSITION 3.13 (SUBSTITUTION). *If $\Delta;\Psi \vdash_{\leq} e':\sigma'$ and $\Delta, x:\sigma', \Delta';\Psi \vdash_{\leq} e:\sigma$, then $\Delta, \Delta';\Psi \vdash_{\leq} [e'/x]e:\sigma$.*

PROPOSITION 3.14 (SUBSTITUTION FOR γ). *If $\Delta;\Psi \cup \{\gamma\} \vdash_{\leq} e:\sigma$, then $\Delta;\Psi \vdash_{\leq} [u/\gamma]e:\sigma$ for any u.*

LEMMA 3.15. *If $\mathcal{D} :: \Delta;\Psi \vdash_{\leq} \lambda x{:}\tau.e:\sigma$ is derivable and $\sigma \leq \sigma_1 \rightarrow \sigma_2$, then $\mathcal{D}$ contains a derivation $\Delta;\Psi \vdash_{\leq} \lambda x{:}\tau.e : \Box(\sigma_1' \rightarrow \sigma_2')$ by* ts_lam *such that $\sigma_1 \leq \sigma_1'$ and $\sigma_2' \leq \sigma_2$.*

PROOF. By induction over the structure of $\mathcal{D}$. □

COROLLARY 3.16. *If $\cdot;\emptyset \vdash_{\leq} \lambda x{:}\tau.e:\sigma_1 \rightarrow \sigma_2$ and $\cdot;\emptyset \vdash_{\leq} e_1:\sigma_1$, then $\cdot;\emptyset \vdash_{\leq} [e_1/x]e:\sigma_2$.*

PROOF. By Lemma 3.15, we have $\cdot;\emptyset \vdash_{\leq} \lambda x{:}\tau.e:\Box(\sigma_1' \rightarrow \sigma_2')$ by ts_lam where $\sigma_1 \leq \sigma_1'$ and $\sigma_2' \leq \sigma_2$. Since $\cdot;\emptyset \vdash_{\leq} e_1:\sigma_1'$, we have $\cdot;\emptyset \vdash_{\leq} [e_1/x]e:\sigma_2'$ by the definition of ts_lam and Proposition 3.13. Then $\cdot;\emptyset \vdash_{\leq} [e_1/x]e:\sigma_2$ follows from $\sigma_2' \leq \sigma_2$ and ts_trans. □

The proof of Lemma 3.17 and Corollary 3.18 is completely analogous.

LEMMA 3.17. *If $\mathcal{D} :: \Delta;\Psi \vdash_{\leq} \int x{:}\tau.e:\sigma$ is derivable and $\sigma \leq \sigma_1 \Rightarrow \sigma_2$, then $\mathcal{D}$ contains a derivation $\Delta;\Psi \vdash_{\leq} \int x{:}\tau.e : \Box(\sigma_1' \Rightarrow \sigma_2')$ by* ts_int *such that $\sigma_1 \leq \sigma_1'$ and $\sigma_2' \leq \sigma_2$.*

COROLLARY 3.18. *If $\cdot;\emptyset \vdash_{\leq} \int x{:}\tau.e:\sigma_1 \Rightarrow \sigma_2$ and $\cdot;\emptyset \vdash_{\leq} v_1:\Box\sigma_1$, then $\cdot;\emptyset \vdash_{\leq} [v_1/x]e:\sigma_2$.*

Then we can prove Theorem 3.12 by induction over the structure of $e \overset{t}{\mapsto}_{\leq} e'$.

The following propositions show the relationship between the two type systems $\vdash_{\gamma}$ and $\vdash_{\leq}$. Proposition 3.19 states that for any expression, its type τ under $\vdash_{\gamma}$ can be recovered from its type σ under $\vdash_{\leq}$, and Proposition 3.20 states that for any expression, its type σ under $\vdash_{\leq}$ refines its type τ under $\vdash_{\gamma}$. The two propositions employ a context refinement judgment $\Delta :: \Gamma$, which extends the refinement judgment $\sigma :: \tau$ with the following rules:

$$\frac{}{\cdot :: \cdot}\ \textsf{ref\_empty}$$

$$\frac{\Delta :: \Gamma \quad \sigma :: \tau}{\Delta, x:\sigma :: \Gamma, x:\tau}\ \textsf{ref\_context}$$

PROPOSITION 3.19. *If $\Delta :: \Gamma$ and $\Delta;\Psi \vdash_{\leq} e:\sigma$, then $\Gamma;\Psi \vdash_{\gamma} e:\tau$ and $\sigma :: \tau$.*

PROPOSITION 3.20. *If $\Delta :: \Gamma$ and $\Gamma;\Psi \vdash_{\gamma} e:\tau$, then $\Delta;\Psi \vdash_{\leq} e:\sigma$ implies $\sigma :: \tau$.*

The following proposition states that for any expression, its type τ under $\vdash_{\gamma}$ is can be thought of as a valid refinement type under $\vdash_{\leq}$.

PROPOSITION 3.21. *If $\Gamma;\Psi \vdash_{\gamma} e:\tau$, then $\Gamma;\Psi \vdash_{\leq} e:\tau$ where types in Γ and τ are all interpreted as refinement types.*

Now Theorem 2.2 follows from Theorem 3.12 in conjunction with Proposition 3.19 and Proposition 3.21.

Soundness of the Refinement Type System

If an expression denotes a regular value, it must consume an empty sampling sequence during its evaluation; otherwise the evaluation result is determined probabilistically and it denotes a probability distribution instead. Since a modal type $\Box\sigma$ denotes a set of regular values, if an expression is assigned a modal type by the type system $\vdash_{\leq}$, it must consume an empty sampling sequence during its evaluation. The following theorem captures this idea formally:

THEOREM 3.22 (SOUNDNESS OF $\vdash_{\leq}$). *If $e \overset{t}{\mapsto}_{\leq} e'$ and $\cdot;\emptyset \vdash_{\leq} e:\Box\sigma$, then $t = \epsilon$.*

The proof of the soundness property requires four lemmas. Note that the proof of Lemma 3.23 exploits Proposition 3.2.

LEMMA 3.23. *For any refinement type σ, there exists no refinement type ρ such that $\mathit{up}(\sigma) \leq \Box\rho$.*

PROOF. By induction on the structure of σ. By Proposition 3.2, cases to consider are $\sigma = \Box\sigma'$ and $\sigma = \sigma_1 \wedge \sigma_2$.

If $\sigma = \Box\sigma'$, then by induction hypothesis, there exists no refinement type ρ such that $\mathit{up}(\sigma') \leq \Box\rho$, that is, $\mathit{up}(\sigma) \leq \Box\rho$.

If $\sigma = \sigma_1 \wedge \sigma_2$ and $up(\sigma) \leq \Box\rho$, then by Proposition 3.2, we have either $up(\sigma_1) \leq \Box\rho$ or $up(\sigma_2) \leq \Box\rho$, which contradicts the induction hypothesis. □

LEMMA 3.24. *If $\cdot;\emptyset \vdash_{\leq} \gamma.e : \sigma$, then there exists no refinement type ρ such that $\sigma \leq \Box\rho$.*

PROOF. By induction over the structure of $\cdot;\emptyset \vdash_{\leq} \gamma.e : \sigma$. □

LEMMA 3.25. *If $\mathcal{D} :: \cdot;\emptyset \vdash_{\leq} e_1\ e_2 : \sigma$ and $\sigma \leq \Box\sigma'$, then $\mathcal{D}$ contains a derivation $\cdot;\emptyset \vdash_{\leq} e_1 : \Box\rho$ for a certain refinement type ρ.*

PROOF. By induction over the structure of $\mathcal{D}$. □

LEMMA 3.26. *If $\mathcal{D} :: \cdot;\emptyset \vdash_{\leq} e_1 \circ e_2 : \sigma$ and $\sigma \leq \Box\sigma'$, then $\mathcal{D}$ contains derivation $\cdot;\emptyset \vdash_{\leq} e_1 : \Box\rho_1$ and $\cdot;\emptyset \vdash_{\leq} e_2 : \Box\rho_2$ for certain refinement types ρ_1 and ρ_2.*

PROOF. By induction over the structure of $\mathcal{D}$. □

PROOF OF THEOREM 3.22. By induction over the structure of $\mathcal{D} :: e \overset{t}{\mapsto}_{\leq} e'$. Cases red_lam, red_int, and red_fix follow vacuously. Case red_dist follows vacuously from Lemma 3.24.

Case: $\mathcal{D} = \dfrac{e_1 \overset{t}{\mapsto} e_1'}{e_1\ e_2 \overset{t}{\mapsto} e_1'\ e_2}$ red_app

If $\cdot;\emptyset \vdash_{\leq} e_1\ e_2 : \Box\sigma$, then there exists σ_1 such that $\cdot;\emptyset \vdash_{\leq} e_1 : \Box\sigma_1$ by Lemma 3.25. By induction hypothesis, we have $t = \epsilon$.
Cases red_itl and red_itr follow from Lemma 3.26 in a similar way. □

4. RELATED WORK

Ramsey and Pfeffer [16] present a stochastic lambda calculus in which any expression denotes a probability distribution. A lambda abstraction translates into a function from regular values to probability distributions. A binary choice construct choose $p\ e_1\ e_2$ represents a linear combination of two probability distributions. The paper presents a denotational semantics based upon the monadic structure of probability distributions and shows how to achieve an elegant implementation of common queries on probability distributions.

Jones [9] presents a metalanguage with a binary choice construct e_1 or$_p$ e_2. The language syntactically distinguishes expressions denoting probability distributions from *function expressions* denoting functions from regular values to probability distributions. In its denotational semantics, a probability distribution is represented by a continuous evaluation, which returns a probability for any element in a lattice of open sets over a topological space.

Koller, McAllester, and Pfeffer [10] present a first order functional language with a coin toss construct flip(p). A program denotes a probability distribution, and its evaluation returns a sample from the probability distribution, although only the top level data constructor is determined. The language employs a lazy evaluation strategy, under which certain probability distributions over infinite domains can be expressed. It is later extended by Pfeffer [13] with higher-order functions and a type system. His language, *IBAL*, generalizes the coin toss construct to a multiple choice construct dist $[p_1 : e_1, \cdots, p_n : e_n]$.

Gupta, Jagadeesan, and Panangaden [7] present *Probabilistic cc*, a stochastic concurrent constraint language. A probabilistic choice construct choose X from Dom in P simulates coin tosses to choose a value for a variable X from a finite set Dom of real numbers. Continuous probability distributions can be expressed with recursion, in the presence of which the language computes the probability distribution for a program as a limit of probability distributions obtained through its finite unwinding.

All the above work essentially employs call-by-value reduction for probability distributions. Note that the lazy evaluation strategy in [10, 13] is analogous to call-by-need reduction in [12], but implements the observational behavior of call-by-value reduction for probability distributions. Saheb-Djahromi [17] recognizes the importance of call-by-name reduction in the formulation of a probabilistic language. The paper presents a probabilistic language based upon LCF, which supports both call-by-value reduction and call-by-name reduction. A term of the form $(p \rightarrow M, q \rightarrow N)$, where $p + q = 1.0$, is employed as a binary choice construct.

Pless and Luger [15] present an extended lambda calculus in which an expression of the form $\sum_i expr_i : p_i$ denotes a probability distribution. The calculus is distinct from all the above work in that any expression denoting a probability distribution always evaluates to a normal form $\sum_i v_i : p_i$, that is, not a certain regular value but the probability distribution itself. To this end, it provides two new reduction rules γ_L and γ_R as well as the β and η reduction rules. The calculus essentially employs call-by-value reduction (γ_R) for probability distributions, although it employs call-by-name reduction (β) for regular values.

All the above work employs probability mass functions over finite domains as their mathematical basis, although they choose different primitive constructs. Any probability distribution over a finite domain can be expressed as long as the probability assigned to each element in the domain is computable. Even an unbiased coin toss construct flip suffices to express such probability distributions because it is equivalent to a binary choice construct [6].

The use of probability mass functions as the mathematical basis implies that discrete probability distributions are the only source of probability distributions. Certain continuous probability distributions (for instance, uniform distributions over real intervals) can be simulated by combining an infinite number of discrete probability distributions, but it may require a special treatment of the underlying calculus (for instance, the lazy evaluation strategy in [10, 13] and the limiting process in [7]). In addition, not every discrete probability distribution can be directly expressed, in particular if its domain is infinite (for instance, geometric distributions).

Thrun [19] extends the imperative language C++ with probabilistic data types, which are created from a template prob<*type*>. The language directly supports continuous probability distributions as well as discrete probability distributions. Continuous probability distributions can be created by predefined functions or by the probloop command, which explicitly maintains dependencies among probability distributions. Our work is an attempt to design a functional probabilistic language with similar goals.

Kozen [11] investigates the semantics of probabilistic while programs. A random assignment $x :=$ random assigns a random number to a variable x and is the source of probability distributions. The language does not assume a particular distribution for the random number generator; it provides only a framework for probabilistic languages. This work is closest to ours in that the source of probability distributions is random numbers generated by the runtime system and that it draws no distinction between discrete and continuous probability distributions.

5. DISCUSSION

5.1 Strengths and Weaknesses of λ_γ

Unified Representation Scheme

Since a sampling construct does not impose any particular semantic restriction on its body, we achieve a unified representation scheme for probability distributions. Discrete probability distributions, continuous probability distributions, and even probability distributions which do not belong to either group are no longer distinguished. Probability domains supported by the calculus are also expanded. For instance, probability distributions over infinite data structures or cyclic domains can be easily specified. Such a unified representation scheme is hard to achieve with other candidates for the mathematical basis for λ_γ such as probability mass functions, probability density functions, or cumulative distribution functions.

Expressiveness

As exemplified in Section 2.4, the calculus provides rich expressiveness to support various types of probability distributions. We can also implement most of the common operations on probability distributions, except the Bayes operation $\sharp$. In the case of continuous probability distributions, for instance, $P \sharp Q$ results in a probability distribution R such that $R(x) = \eta P(x)Q(x)$ where η is a normalization constant. The Bayes operation is hard to implement even if we can compute probability density functions $P(x)$ and $Q(x)$ exactly because the integration of $P(x)Q(x)$ over a probability domain, which is necessary to obtain η, may not be feasible. However, if we have an expression e_P denoting P and a function $q(x)$ such that $q(x) = kQ(x) \le c$ for certain constants k and c (or vice versa), we can implement the Bayes operation by the rejection method:

```
let rec $bayes = \lambda e_p\!:\!\tau.\textstyle\int q\!:\!\tau \Rightarrow \mathsf{real}.\int c\!:\!\mathsf{real}.$
    sample $x$ from $e_p$ in
    sample $u$ from $\gamma.\gamma$ in
    if $u < (q \circ x)/c$ then
        $x$
    else
        $bayes\ e_p \circ q \circ c$
```

This is a reasonable assumption which is often satisfied in practice. Furthermore we can dispense with the constant c if we approximate sampling functions with a set of weighted samples, which is known as the sample-based representation for probability distributions [5].

Lack of Support for Precise Reasoning

The quintessential weakness of λ_γ is that it does not permit a precise implementation of queries on probability distributions such as expectation. In other words, even if we can express a probability distribution of interest, we may not be able to reason about it. For instance, we may often wish to compute the mean of a probability distribution in order to choose a representative sample, which can be accomplished by computing its expectation with respect to an identity function. However it is hard to answer an expectation query because, in general, we cannot convert a sampling function to a probability mass function (or a probability density function).

Still we believe that there exists a practical solution to this problem. For discrete probability distributions over finite domains, we plan to investigate intensional code analysis to produce accurate answers. For all other kinds of probability distributions, we can obtain approximate answers by generating a number of samples and then analyzing them. For instance, we can employ the Monte Carlo simulation method to answer an expectation query on any probability distribution.

Lack of support for precise reasoning is the price we pay for rich expressiveness provided by the calculus. We believe that the benefit of basing the calculus upon sampling functions far outweighs this inevitable disadvantage because precise reasoning on general probability distributions is out of the question at any rate.

5.2 Utility of the Refinement Type System

Since the type system $\vdash_\le$ is capable of distinguishing expressions denoting regular values from expressions denoting probability distributions, its potential usefulness is commensurate with the overhead of probabilistic computation: the higher the overhead is, the more cost we can save in evaluating expressions denoting regular values. In the operational semantics of λ_γ, the overhead of probabilistic computation is minimal because every expression evaluates to a regular value and hence we never need to manipulate probability distributions directly. In fact, random numbers generated by the runtime system are the only overhead, which is intrinsic to λ_γ. Thus we cannot benefit from $\vdash_\le$ in a naive implementation of the operational semantics of λ_γ.

However, if we evaluate an expression multiple times (for instance, to obtain an approximate answer to a query on a probability distribution), we can cache the result of evaluating any subexpression denoting a regular value and reuse it in subsequent evaluations. In this case, we can exploit $\vdash_\le$ to mark such subexpressions and avoid evaluating them more than once. Also, if we extend the operational semantics of λ_γ in such a way that an expression evaluates not to a regular value but to an object representing a probability distribution (for instance, a sampling function in the form of closure) or its approximation (for instance, a set of samples), the overhead of probabilistic computation increases, and there is a good chance of exploiting $\vdash_\le$ in an implementation of the new operational semantics.

Apart from its usefulness in implementing λ_γ, the type system $\vdash_\le$ is valuable in itself as a means of verifying a specific property of a given expression, that is, whether it denotes a regular value or not. This is because an objective in developing a type system is to provide programmers with more information on expressions, which is achieved in the case of $\vdash_\le$.

6. CONCLUSION AND FUTURE WORK

We have preliminary evidence that a probabilistic language based upon the calculus λ_γ is viable in applications involving massive probabilistic computation. We have experimented with the robot localization problem both by simulation and on a real robot. The pose of a robot is estimated with a continuous probability distribution over three-dimensional real space.

At its core, our implementation employs a Bayes filter [8]:

$$\begin{array}{lcll} Bel(p) & \leftarrow & \eta P(o|p) Bel(p) & (1) \\ Bel(p) & \leftarrow & \int_{p'} A(p|a,p') Bel(p') dp' & (2) \end{array}$$

$Bel(p)$ is the probability for a pose p. $P(o|p)$ is the probability that the robot at a pose p observes an observation o, and η is a normalization constant. $A(p|a,p')$ is the probability that the robot is at a pose p after taking an action a at another pose p'.

In our implementation, Bel is represented with a sampling function. Although we cannot accurately compute $P(o|p)$ given an observation o, we can still implement the formula (1) with the Bayes operation as explained in Section 5.1. To this end, we approximate Bel with a set of samples whenever we update Bel by (1). The formula (2) is easily implemented as an integration. Our findings indicate that sampling functions provide a practical solution to the problem of representing complex probability distributions without sacrificing too much accuracy; although we lose accuracy in representing Bel by approximating it with a set of samples, our implementation successfully localizes the robot in real time.

It is important to note that expressiveness of a probabilistic language does not necessarily indicate its practicality, which is usually determined by the degree of support for reasoning on probability distributions. For instance, we can easily encode a Bayesian network with discrete nodes in λ_γ, but we may not be able to accurately answer any interesting query on it. In this particular problem, a better choice is a probabilistic language which may not be as expressive as λ_γ but supports precise reasoning on discrete probability distributions. On the other hand, if we are interested in encoding a Bayesian network with continuous nodes, λ_γ can be considered to be a good choice because it facilitates the encoding of such a network and also provides a practical means of answering queries on it. Therefore an important consideration in choosing a probabilistic language is to answer *"what kind of probability distributions do we want to compute?"*. We believe that for those problems in which precise reasoning on probability distributions of interest is impossible in practice or not necessary (for instance, the robot localization problem), λ_γ is a good choice because of its rich expressiveness and support for approximate reasoning.

We are investigating the possibility of revising the refinement type system $\vdash_\leq$ into a more elegant form because it appears to be *ad hoc*. We are also extending the operational semantics of λ_γ as suggested in Section 5.2. We plan to implement a probabilistic language based upon λ_γ extended with effects and polymorphism.

Acknowledgment

I am grateful to Frank Pfenning, who guided me through this research and suggested many key ideas in this paper. I thank Jonathan Moody and Aleks Nanevski for their helpful discussion, Sebastian Thrun and Mike Montemerlo for setting up the experiment environment, and three anonymous reviewers for their insightful suggestions. This research has been sponsored by DARPA's MARS Program (contracts N66001-01-C-6018 and NBCH1020014), DARPA's CoABS Program (contract F30602-98-2-0137), and DARPA's MICA Program (contract F30602-01-C-0219), all of which are gratefully acknowledged.

7. REFERENCES

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

[2] R. Davies and F. Pfenning. Intersection types and computational effects. In *ICFP-00*, pages 198–208. ACM Press, 2000.

[3] R. Davies and F. Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3), 2001.

[4] A. Edalat and P. J. Potts. A new representation for exact real numbers. In *Electronic Notes in Theoretical Computer Science*, volume 6. Elsevier Science Publishers, 2000.

[5] D. Fox, W. Burgard, F. Dellaert, and S. Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *AAAI-99*, pages 343–349. AAAI/MIT Press, 1999.

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

[7] V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *26th ACM POPL*, pages 189–202. ACM Press, 1999.

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

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

[10] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *AAAI-97/IAAI-97*, pages 740–747. AAAI Press, 1997.

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

[12] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.

[13] A. Pfeffer. IBAL: A probabilistic rational programming language. In *IJCAI-01*, pages 733–740. Morgan Kaufmann Publishers, 2001.

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

[15] D. Pless and G. Luger. Toward general analysis of recursive probability models. In *UAI-01*, pages 429–436. Morgan Kaufmann Publishers, 2001.

[16] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *29th ACM POPL*, pages 154–165. ACM Press, 2002.

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

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

[19] S. Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *ICRA-00*. IEEE, 2000.