From Bayesian Notation to Pure Racket
via Discrete Measure-Theoretic Probability in $\lambda_{\mathbf{ZFC}}$

Neil Toronto and Jay McCarthy

PLT @ Brigham Young University, Provo, Utah, USA
neil.toronto@gmail.com and jay@cs.byu.edu

Abstract. Bayesian practitioners build models of the world without regarding how difficult it will be to answer questions about them. When answering questions, they put off approximating as long as possible, and usually must write programs to compute converging approximations. Writing the programs is distracting, tedious and error-prone, and we wish to relieve them of it by providing languages and compilers.
Their style constrains our work: the tools we provide cannot approximate early. Our approach to meeting this constraint is to 1) determine their notation's meaning in a suitable theoretical framework; 2) generalize our interpretation in an uncomputable, *exact* semantics; 3) *approximate* the exact semantics and prove convergence; and 4) implement the approximating semantics in Racket (formerly PLT Scheme). In this way, we define languages with at least as much exactness as Bayesian practitioners have in mind, and also put off approximating as long as possible.
In this paper, we demonstrate the approach using our preliminary work on discrete (countably infinite) Bayesian models.

Keywords: Semantics, Domain-specific languages, Probability theory

1 Introduction

Bayesian practitioners define *models*, or probabilistic relationships among objects of study, without regard to whether future calculations are closed-form or tractable. They are loath to make simplifying assumptions. (If some probabilistic phenomenon is best described by an unsolvable integral or infinitely many distributions, so be it.) When they must approximate, they often create two models: an "ideal" model first, and a second model that approximates it.

Because they create models without regard to future calculations, they usually must accept approximate answers to queries about them. Typically, they adapt algorithms that compute converging approximations in programming languages they are familiar with. The process is tedious and error-prone, and involves much performance tuning and manual optimization. It is by far the most time-consuming part of their work—and also the most automatable part.

They follow this process to adhere to an overriding philosophy: an approximate answer to the right question is worth more than an exact answer to an approximate question. Thus, they put off approximating as long as possible.

We must also adhere to this philosophy because Bayesian practitioners are unlikely to use a language that requires them to approximate early, or that approximates earlier than they would. We have found that a good way to put the philosophy into practice in language design is to create two semantics: an "ideal," or *exact* semantics first, and a converging, *approximating* semantics.

1.1 Theory of Probability

Measure-theoretic probability is the most successful theory of probability in precision, maturity, and explanatory power. In particular, it explains every Bayesian model. We therefore define the exact semantics as a transformation from Bayesian notation to measure-theoretic calculations.

Measure theory treats finite, countably infinite, and uncountably infinite probabilistic outcomes uniformly, but with significant complexity. Though there are relatively few important Bayesian models that require countably many outcomes but not uncountably many, in our preliminary work, we deal with only countable sets. This choice avoids most of measure theory's complexity while retaining its functional structure, and still requires approximation.

1.2 Approach and Target Language

For three categories of Bayesian notation, we

1. Manually interpret an unambiguous subclass of typical notation.
2. Mechanize the interpretation with a semantic function.
3. If necessary, create an approximation and prove convergence.
4. Implement the approximation in Racket [3] (formerly PLT Scheme).

This approach is most effective if the target language can express measure-theoretic calculations and is similar to Racket in structure and semantics.

To this end, we are concurrently developing λ_{ZFC}: an untyped, call-by-value lambda calculus extended with sets as values, and primitive set operators that correspond with the Zermelo-Fraenkel axioms and Choice. Mixing lambdas and sets has been done in automated theorem proving [17, 18, 5]; it is possible to define exact real arithmetic, and easy to express infinite set operations and infinite sums. For this paper, we assume those operations are already defined. We freely use syntactic sugar like infix, summation, set comprehensions, and pattern-matching definitions. In short, we intend λ_{ZFC} to be contemporary mathematics with well-defined lambdas, or a practical lambda calculus with infinite sets.

For example, ***image*** is a `map`-like primitive set operator corresponding to the replacement axiom schema. If f is a lambda and A is a set, ***image*** $f\ A\ =\ \{f\ x \mid x \in A\}$ applies f to every element of A and returns the set of results.

Besides lambdas, λ_{ZFC} has an another class of applicable values: set-theoretic functions, or **mappings**. A mapping is just a set of pairs (x, y) where each x is unique. If g is a mapping and x is in its domain, $g\ x$ returns the y for which $(x, y) \in g$. Restricting a lambda f to a domain A returns a mapping:

$$f\big|_A\ =\ \{(x, f\ x) \mid x \in A\}\ =\ \textbf{\textit{image}}\ (\lambda x.\,(x, f\ x))\ A \tag{1}$$

Mappings can also be restricted using (1). We often write mappings as $\lambda(x \in A).e$ instead of $(\lambda x.e)\big|_A$. We think of (A, f) as a *lazy* mapping.

Though λ_{ZFC} has no formal type system, we find it helpful to reason about types informally. When we do, $A \Rightarrow B$ is a lambda or mapping type, $A \to B$ is the set of total mappings from A to B, and a set is a membership proposition.

1.3 Bayesian Languages

The Bayesian notation we interpret falls into three categories:

1. **Expressions**, which have no side effects, with $\mathcal{R}[\![\cdot]\!] : \lambda_{\text{ZFC}} \Rightarrow \lambda_{\text{ZFC}}$.
2. **Queries**, which observe side effects, with $\mathbf{P}[\![\cdot]\!], \mathbf{D}[\![\cdot]\!] : \textit{propositions} \Rightarrow \lambda_{\text{ZFC}}$.
3. **Statements**, which create side effects, with $\mathcal{M}[\![\cdot]\!] : \textit{statements} \Rightarrow \lambda_{\text{ZFC}}$.

We use λ_{ZFC} as a term language for all of our mathematics. We write Bayesian notation in sans serif, Racket in `fixed width`, common keywords in **bold** and invented keywords in ***bold italics***. We omit proofs for space.

2 The Expression Language

2.1 Background Theory: Random Variables

Most practitioners of probability understand random variables as free variables whose values have ambient probabilities. But measure-theoretic probability defines a **random variable** X as a total mapping

$$X : \Omega \to S_X \tag{2}$$

where Ω and S_X are sets called **sample spaces**, with elements called **outcomes**. Random variables define and limit what is observable about any outcome $\omega \in \Omega$, so we call outcomes in S_X ***observable outcomes***.

Example 1. Suppose we want to encode, as a random variable E, the act of observing whether the outcome of a die roll is even or odd.

A complicated way is to define Ω as the possible states of the universe. $E : \Omega \to \{even, odd\}$ must simulate the universe until the die is still, and then recognize the outcome. Hopefully, the probability that $E\ \omega = even$ is $\frac{1}{2}$.

A tractable way defines $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $E\ \omega = even$ if $\omega \in \{2, 4, 6\}$, otherwise odd. The probability that $E\ \omega = even$ is the sum of probabilities of every even $\omega \in \Omega$, or $\frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$.

If we are interested in observing only evenness, we can define $\Omega = \{even, odd\}$, each with probability $\frac{1}{2}$, and $E\ \omega = \omega$. □

Random variables enable a kind of probabilistic abstraction. The example does it twice. The first makes calculating the probability that $E\ \omega = even$ tractable. The second is an optimization. In fact, redefining Ω, the random variables, and the probabilities of outcomes—without changing the probabilities of *observable* outcomes—is the essence of measure-theoretic optimization.

Defining random variables as functions is also a good factorization: it separates nondeterminism from assigning probabilities. It allows us to interpret expressions involving random variables without considering probabilities at all.

$$\mathcal{R}[\![X]\!] = X \qquad \mathcal{R}[\![x]\!] = \textbf{\textit{pure}}\ x \qquad \mathcal{R}[\![v]\!] = \textbf{\textit{pure}}\ v$$
$$\mathcal{R}[\![(e_f\ e_1\ \ldots\ e_n)]\!] = \textbf{\textit{ap}}^*\ \mathcal{R}[\![e_f]\!]\ \mathcal{R}[\![e_1]\!]\ \ldots\ \mathcal{R}[\![e_n]\!]$$
$$\mathcal{R}[\![\lambda x_1 \ldots x_n.e]\!] = \lambda\omega.\lambda x_1 \ldots x_n.\,(\mathcal{R}[\![e]\!]\ \omega)$$
$$\textbf{\textit{pure}}\ c = \lambda\omega.c, \quad \textbf{\textit{ap}}^*\ F\ X_1\ \ldots\ X_n = \lambda\omega.\,((F\ \omega)\ (X_1\ \omega)\ \ldots\ (X_n\ \omega))$$

Fig. 1. Random variable expression semantics. The source and target language are both λ_{ZFC}. Conditionals and primitive operators are trivial special cases of application.

2.2 Interpreting Random Variable Expressions As Computations

When random variables are regarded as free variables, arithmetic with random variables is no different from deterministic arithmetic. Measure-theoretic probability uses the same notation, but regards it as implicit pointwise lifting (as in vector arithmetic). For example, if A, B and C are random variables, $\mathsf{C} = \mathsf{A} + \mathsf{B}$ means $C\ \omega = (A\ \omega) + (B\ \omega)$, and $\mathsf{B} = \mathsf{4} + \mathsf{A}$ means $B\ \omega = 4 + (A\ \omega)$.

Because we write all of our math in λ_{ZFC}, we can extend the class of random variables from $\Omega \to S_X$ to $\Omega \Rightarrow S_X$. Including lambdas as well as mappings makes it easy to interpret unnamed random variables: $\mathsf{4} + \mathsf{A}$, or $(+\ \mathsf{4}\ \mathsf{A})$, means $\lambda\omega.\,(+\ 4\ (A\ \omega))$. Lifting constants allows us to interpret expressions uniformly: if we interpret $+$ as $\mathit{Plus} = \lambda\omega.+$ and $\mathsf{4}$ as $\mathit{Four} = \lambda\omega.4$, then $(+\ \mathsf{4}\ \mathsf{A})$ means

$$\lambda\omega.\,((\mathit{Plus}\ \omega)\ (\mathit{Four}\ \omega)\ (A\ \omega)) \tag{3}$$

We abstract lifting and application with the combinators

$$\begin{aligned} \textbf{\textit{pure}}\ c\ &=\ \lambda\omega.c \\ \textbf{\textit{ap}}^*\ F\ X_1\ \ldots\ X_n\ &=\ \lambda\omega.\,((F\ \omega)\ (X_1\ \omega)\ \ldots\ (X_n\ \omega)) \end{aligned} \tag{4}$$

so that $(+\ \mathsf{4}\ \mathsf{A})$ means $\textbf{\textit{ap}}^*\ (\textbf{\textit{pure}}\ +)\ (\textbf{\textit{pure}}\ 4)\ A = \cdots = \lambda\omega.\,(+\ 4\ (A\ \omega))$. These combinators define an **idiom** [13], which is like a monad but can impose a partial order on computations. Our ***random variable idiom*** instantiates the environment idiom with the type constructor $(I\ a) = (\Omega \Rightarrow a)$ for some Ω.

$\mathcal{R}[\![\cdot]\!]$ (Fig. 1), the semantic function that interprets random variable expressions, targets this idiom. It does mechanically what we have done manually, and additionally interprets lambdas. For simplicity, it follows probability convention by assuming single uppercase letters are random variables. Fig. 1 assumes syntactic sugar has been replaced; e.g. that application is in prefix form.

$\mathcal{R}[\![\cdot]\!]$ may return lambdas that do not converge when applied. These lambdas do not represent random variables, which are total. We will be able to recover mappings by restricting them, as in $\mathcal{R}[\![(+\ \mathsf{4}\ \mathsf{A})]\!]\big|_{\Omega}$.

2.3 Implementation in Racket

Figure 2 shows `RV` and a snippet of `RV/kernel`, the macros that implement $\mathcal{R}[\![\cdot]\!]$. `RV` fully expands expressions into Racket's kernel language, allowing `RV/kernel` to

```
(define-syntax (RV/kernel stx)
  (syntax-parse stx
    [(_ Xs:ids e:expr)
     (syntax-parse #'e #:literal-sets (kernel-literals)
       [X:id  #:when (free-id-in? #'Xs #'X)  #'X]
       [x:id        #'(pure x)]
       [(quote c)   #'(pure (quote c))]
       [(%#plain-app e ...)  #'(ap* (RV/kernel Xs e) ...)]
       ....)]))
(define-syntax (RV stx)
  (syntax-parse stx
    [(_ Xs:ids e:expr)
     #`(RV/kernel Xs #,(local-expand #'e 'expression empty))]))
```

Fig. 2. A well-embedded implementation of $\mathcal{R}[\![\cdot]\!]$.

transform any pure Racket expression into a random variable. Both use Racket's new `syntax-parse` library [2]. `RV/kernel` raises a syntax error on `set!`, but there is no way to disallow applying functions that have effects.

Rather than differentiate between kinds of identifiers, `RV` takes a list of known random variable identifiers as an additional argument. It wraps other identifiers with `pure`, allowing arbitrary Racket values to be random variables.

3 The Query Language

It is best to regard statements in Bayesian notation as specifications for the results of later observations. We therefore interpret queries before interpreting statements. First, however, we must define the state objects that queries observe.

3.1 Background Theory: Probability Spaces

In practice, functions called **distributions** assign probabilities or probability densities to observable outcomes. Practitioners state distributions for certain random variables, and then calculate the distributions of others.

Measure-theoretic probability generalizes assigning probabilities and densities using **probability measures**, which assign probabilities to *sets* of outcomes. There are typically no special random variables: all random variable distributions are calculated from one global probability measure.

It is generally not possible to assign meaningful probabilities to all subsets of a sample space Ω—except when Ω is countable. We thus deal here with **discrete probability measures** $\mathbb{P} : \mathcal{P}(\Omega) \to [0, 1]$. Any discrete probability measure is uniquely determined by its value on singleton sets, or by a **probability mass function** $P : \Omega \to [0, 1]$. It is easy to convert P to a probability measure:

$$\boldsymbol{sum}\ P\ A = \sum_{\omega \in A} P\ \omega \tag{5}$$

Then $\mathbb{P} = \textbf{\textit{sum}}\ P$. Converting the other direction is also easy: $P\ e = \mathbb{P}\ \{e\}$.

A **discrete probability space** (Ω, P) embodies all probabilistic nondeterminism introduced by statements. It is fine to think of Ω as the set of all possible states of a write-once memory, with P assigning a probability to each state.

3.2 Background Theory: Queries

Any probability can be calculated from (Ω, P). For example, suppose we want to calculate, as in Example 1, the probability of an even die outcome. We must apply $\mathbb{P}$ to the correct subset of Ω. Suppose that $\Omega = \{1,2,3,4,5,6\}$ and that $P = [1,2,3,4,5,6 \to \frac{1}{6}]$ determines $\mathbb{P}$. The probability that E outputs *even* is

$$\mathbb{P}\ \{\omega \in \Omega \mid E\ \omega = \mathit{even}\}\ =\ \mathbb{P}\ \{2,4,6\}\ =\ \textbf{\textit{sum}}\ P\ \{2,4,6\}\ =\ \tfrac{1}{2} \tag{6}$$

This is a **probability query**.

Alternatively, we could use a **distribution query** to calculate E's distribution $\mathbb{P}_E$, and then apply it to $\{\mathit{even}\}$. Measure-theoretic probability elegantly defines $\mathbb{P}_E$ as $\mathbb{P} \circ E^{-1}$, but for now we do not need a measure. We only need the probability mass function $P_E\ e = \textbf{\textit{sum}}\ P\ \left(E^{-1}\ \{e\}\right)$. Applying it yields

$$P_E\ \mathit{even}\ =\ \textbf{\textit{sum}}\ P\ \left(E^{-1}\ \{\mathit{even}\}\right)\ =\ \textbf{\textit{sum}}\ P\ \{2,4,6\}\ =\ \tfrac{1}{2} \tag{7}$$

More abstractly, we can calculate discrete distribution queries using

$$\textbf{\textit{dist}}\ X\ (\Omega, P)\ =\ \lambda(x \in S_X).\textbf{\textit{sum}}\ P\ \left(X\big|_{\Omega}^{-1}\ \{x\}\right) \tag{8}$$

where $S_X = \textbf{\textit{image}}\ X\ \Omega$. Recall that $X\big|_{\Omega}$ converts X, which may be a lambda, to a mapping with domain Ω, on which preimages are well-defined.

3.3 Interpreting Query Notation

When random variables are regarded as free variables, special notation $\mathsf{P}[\cdot]$ replaces applying $\mathbb{P}$ and sets become propositions. For example, a common way to write "the probability of an even die outcome" in practice is $\mathsf{P}[\mathsf{E} = \mathsf{even}]$.

The semantic function $\mathcal{R}[\![\cdot]\!]$ turns propositions about random variables into predicates on Ω. The set corresponding to the proposition is the preimage of $\{\mathit{true}\}$. For $\mathsf{E} = \mathsf{even}$, for example, it is $\mathcal{R}[\![\mathsf{E} = \mathsf{even}]\!]\big|_{\Omega}^{-1}\ \{\mathit{true}\}$. In general,

$$\textbf{\textit{sum}}\ P\ \left(\mathcal{R}[\![e]\!]\big|_{\Omega}^{-1}\ \{\mathit{true}\}\right)\ =\ \textbf{\textit{dist}}\ \mathcal{R}[\![e]\!]\ (\Omega, P)\ \mathit{true} \tag{9}$$

calculates $\mathsf{P}[e]$ when e is a proposition; i.e. when $\mathcal{R}[\![e]\!] : \Omega \Rightarrow \{\mathit{true}, \mathit{false}\}$.

Although probability queries have common notation, there seems to be no common notation that denotes distributions *per se*. The typical workarounds are to write implicit formulas like $\mathsf{P}[\mathsf{E} = \mathsf{e}]$ and to give distributions suggestive names like P_E. Some theorists use $\mathcal{L}[\cdot]$, with $\mathcal{L}$ for *law*, an obscure synonym of *distribution*. We define $\mathbf{D}[\![\cdot]\!]$ in place of $\mathcal{L}[\cdot]$. Then $\mathbf{D}[\![\mathsf{E}]\!]$ denotes E's distribution.

Though we could define semantic functions $\mathbf{P}[\![\cdot]\!]$ and $\mathbf{D}[\![\cdot]\!]$ right now, we are putting them off until after interpreting statements.

```
(struct mapping (domain proc)
  #:property prop:procedure (λ (f x) ((mapping-proc f) x)))
(struct fmapping (default hash)
  #:property prop:procedure
  (λ (f x) (hash-ref (fmapping-hash f) x (fmapping-default f))))

(define appx-z (make-parameter +inf.0))
(define (finitize ps)
  (match-let* ([(mapping Ω P)  ps]
               [Ωn  (cotake Ω (appx-z))]
               [qn  (apply + (map P Ωn))])
    (mapping Ωn (λ (ω) (/ (P ω) qn)))))

(define ((dist X) ps)
  (match-define (mapping Ω P) ps)
  (fmapping 0 (for/fold ([h  (hash)]) ([ω  (in-list Ω)])
                (hash-set h (X ω) (+ (P ω) (hash-ref h (X ω) 0))))))
```

Fig. 3. Implementation of finite approximation and distribution queries in Racket.

3.4 Approximating Queries

Probabilities are real numbers. They remain real in the approximating semantics; we use floating-point approximation and exact rationals in the implementation.

Arbitrary countable sets are not finitely representable. In the approximating semantics, we restrict Ω to recursively enumerable sets. The implementation encodes them as lazy lists. We trust users to not create "sets" with duplicates.

When A is infinite, ***sum*** P A is an infinite series. With A as a lazy list, it is easy to compute a converging approximation—but then approximate answers to distribution queries sum to values less than 1. Instead, we approximate Ω and normalize P, which makes the sum finite and the distributions proper.

Suppose $(\omega_1, \omega_2, \ldots)$ is an enumeration of Ω. Let $z \in \mathbb{N}$ be the length of the prefix $\Omega_z = \{\omega_1, \ldots, \omega_z\}$ and let $P_z\ \omega = (P\ \omega)/(\textbf{\textit{sum}}\ P\ \Omega_z)$. Then P_z converges to P. We define ***finitize*** $(\Omega, P) = (\Omega_z, P_z)$ with $z \in \mathbb{N}$ as a free variable.

3.5 Implementation in Racket

Fig. 3 shows the implementations of ***finitize*** and ***dist*** in Racket. The free variable z appears as a *parameter* `appx-z`: a variable with static scope but dynamic extent. The `cotake` procedure returns the prefix of a lazy list as a finite list.

To implement ***dist***, we need to represent mappings in Racket. The applicable struct type `mapping` represents lazy mappings with possibly infinite domains. A `mapping` named `f` can be applied with `(f x)`. We do not ensure `x` is in the domain because checking is semidecidable and nontermination is a terrible error message. For distributions, checking is not important; the observable domain is.

However, we do not want `dist` to return lazy mappings. Doing so is inefficient: every application of the mapping would filter `Ω`. Further, `dist` always receives a `finitize`d probability space. We therefore define `fmapping` for mappings that are constant on all but a finite set. For these values, `dist` builds a hash table by computing the probabilities of all preimages in one pass through `Ω`.

We do use `mapping`, but only for probability spaces and stated distributions.

4 Conditional Queries

For Bayesian practitioners, the most meaningful queries are **conditional** queries: those *conditioned on*, or *given*, some random variable's value. (For example, the probability an email is spam given it contains words like "madam," or the distribution over suspects given security footage.) A language without conditional queries is of little more use to them than a general-purpose language.

Measure-theoretic conditional probability is too involved to accurately summarize here. When $\mathbb{P}$ is discrete, however, the conditional probability of set A given set B (i.e. asserting that $\omega \in B$), simplifies to

$$\mathsf{P}[A \mid B] \;=\; (\mathbb{P}\ A \cap B)/(\mathbb{P}\ B) \tag{10}$$

In theory and practice, $\mathsf{P}[\cdot \mid \cdot]$ is special notation. As with $\mathsf{P}[\cdot]$, practitioners apply it to propositions. They define it with $\mathsf{P}[e_A \mid e_B] = \mathsf{P}[e_A \wedge e_B]/\mathsf{P}[e_B]$.

Example 2. Extend Example 1 with random variable L $\omega = \mathit{low}$ if $\omega \leq 3$, else high. The probability that $\mathsf{E} = \mathsf{even}$ given $\mathsf{L} = \mathsf{low}$ is

$$\mathsf{P}[\mathsf{E} = \mathsf{even} \mid \mathsf{L} = \mathsf{low}] \;=\; \frac{\mathsf{P}[\mathsf{E} = \mathsf{even} \wedge \mathsf{L} = \mathsf{low}]}{\mathsf{P}[\mathsf{L} = \mathsf{low}]} \;=\; \frac{\sum\limits_{\omega \in \{2\}} P\ \omega}{\sum\limits_{\omega \in \{1,2,3\}} P\ \omega} \;=\; \frac{\frac{1}{6}}{\frac{1}{2}} \;=\; \frac{1}{3} \tag{11}$$

Less precisely, there are proportionally fewer even outcomes when $\mathsf{L} = \mathsf{low}$. □

Conditional *distribution* queries ask how one random variable's output influences the distribution of another. As with unconditional distribution queries, practitioners work around a lack of common notation. For example, they might write the distribution of E given L as $\mathsf{P}[\mathsf{E} = \mathsf{e} \mid \mathsf{L} = \mathsf{l}]$ or $\mathsf{P}_{\mathsf{E}|\mathsf{L}}$.

It is tempting to define $\mathbf{P}[\![\cdot \mid \cdot]\!]$ in terms of $\mathbf{P}[\![\cdot]\!]$ (and $\mathbf{D}[\![\cdot \mid \cdot]\!]$ in terms of $\mathbf{D}[\![\cdot]\!]$). However, defining conditioning as an operation on probability spaces instead of on queries is more flexible, and it better matches the unsimplified measure theory. The following abstraction returns a discrete probability space in which Ω is restricted to the subset where random variable Y returns y:

$$\begin{aligned} \textbf{\textit{cond}}\ Y\ y\ (\Omega, P) \;=\; (\Omega', P')\ \text{ where }\ & \Omega' = Y\big|_{\Omega}^{-1}\ \{y\} \\ & P' = \lambda(\omega \in \Omega').(P\ \omega)/(\textbf{\textit{sum}}\ P\ \Omega') \end{aligned} \tag{12}$$

Then $\mathsf{P}[\mathsf{E} = \mathsf{even} \mid \mathsf{L} = \mathsf{low}]$ means $\textbf{\textit{dist}}\ E\ (\textbf{\textit{cond}}\ L\ \mathit{low}\ (\Omega, P))\ \mathit{even}$.

We approximate ***cond*** by applying ***finitize*** to the probability space. Its implementation uses finite list procedures instead of set operators.

5 The Statement Language

Random variables influence each other through global probability spaces. However, because practitioners regard random variables as free variables instead of as functions of a probability space, they state facts about random variable distributions instead of facts about probability spaces. Though they call such collections of statements *models*,[1] to us they are ***probabilistic theories***. A ***model*** is a probability space and random variables that imply the stated facts.

Discrete ***conditional theories*** can always be written to conform to

$$t_i \ ::= \ X_i \sim e_i;\ t_{i+1} \mid X_i := e_i;\ t_{i+1} \mid e_a = e_b;\ t_{i+1} \mid \epsilon \tag{13}$$

Further, they can always be made ***well-formed***: an e_j may refer to some X_i only when $j > i$ (i.e. no circular bindings). We start by interpreting the most common kind of Bayesian theories, which contain only distribution statements.

5.1 Interpreting Common Conditional Theories

Example 3. Suppose we want to know only whether a die outcome is even or odd, high or low. If L's distribution is $P_L = [low, high \mapsto \frac{1}{2}]$, then E's distribution depends on L's output.

Define $P_{E|L} : S_L \to S_E \to [0,1]$ by $P_{E|L}\ low = [even \mapsto \frac{1}{3}, odd \mapsto \frac{2}{3}]$ and $P_{E|L}\ high = [even \mapsto \frac{2}{3}, odd \mapsto \frac{1}{3}]$.[2] The conditional theory could be written

$$\mathsf{L} \sim \mathsf{P_L};\ \mathsf{E} \sim \left(\mathsf{P_{E|L}}\ \mathsf{L}\right) \tag{14}$$

If L is a measure-theoretic random variable, $\left(\mathsf{P_{E|L}}\ \mathsf{L}\right)$ does not type-check: $L : \Omega \to S_L$ is clearly not in S_L. The *intent* is that E's distribution depends on L, and that $P_{E|L}$ specifies how. □

We can regard $\mathsf{L} \sim \mathsf{P_L}$ as a constraint: for every model (Ω, P, L), ***dist*** $L\ (\Omega, P)$ must be P_L. Similarly, $\mathsf{E} \sim \left(\mathsf{P_{E|L}}\ \mathsf{L}\right)$ means E's conditional distribution is $P_{E|L}$. We have been using the model $\Omega = \{1,2,3,4,5,6\}$, $P = [1,2,3,4,5,6 \mapsto \frac{1}{6}]$, and the obvious E and L. It is not hard to verify that this is also a model:

$$\begin{aligned} &\Omega = \{low, high\} \times \{even, odd\} \qquad L\ \omega = \omega_1 \qquad E\ \omega = \omega_2 \\ &P = [(low, even), (high, odd) \mapsto \tfrac{1}{6}, (low, odd), (high, even) \mapsto \tfrac{2}{6}] \end{aligned} \tag{15}$$

The construction of Ω, L and E in (15) clearly generalizes, but P is trickier. Fully justifying the generalization (including that it meets implicit independence assumptions that we have not mentioned) is rather tedious, so we do not do it here. But, for the present example, it is not hard to check these facts:

$$\begin{aligned} P\ \omega &= (P_L\ (L\ \omega)) \times \left(P_{E|L}\ (L\ \omega)\ (E\ \omega)\right) \\ \text{or} \quad P &= \mathcal{R}[\![(P_L\ L) \times \left(\left(P_{E|L}\ L\right)\ E\right)]\!] \end{aligned} \tag{16}$$

[1] In the colloquial sense, probably to emphasize their essential incompleteness.

[2] Usually, $P_{E|L} : S_E \times S_L \to [0,1]$. We reorder and curry to simplify interpretation.

$$\boldsymbol{dist}_{ps}\ X\ (\Omega, P) = (\Omega, P, P_X) \quad \text{where} \quad S_X = \boldsymbol{image}\ X\ \Omega$$
$$P_X = \lambda(x \in S_X).\boldsymbol{sum}\ P\ \left(X\big|_{\Omega}^{-1}\ \{x\}\right)$$
$$\boldsymbol{cond}_{ps}\ Y\ y\ (\Omega, P) = \left(\Omega', P', \_\right) \quad \text{where} \quad \Omega' = Y\big|_{\Omega}^{-1}\ \{y\}$$
$$P' = \lambda(\omega \in \Omega').(P\ \omega)/\left(\boldsymbol{sum}\ P\ \Omega'\right)$$
$$\boldsymbol{extend}_{ps}\ K_i\ (\Omega_{i-1}, P_{i-1}) = (\Omega_i, P_i, X_i)$$
$$\text{where}\ \ S_i'\ \omega = \boldsymbol{domain}\ (K_i\ \omega), \quad \Omega_i = (\omega \in \Omega_{i-1}) \times \left(S_i'\ \omega\right)$$
$$X_i\ \omega = \omega_j\ \ (\text{where } j = \text{length of any } \omega \in \Omega_{i-1}), \quad P_i = \mathcal{R}[\![P_{i-1} \times (K_i\ X_i)]\!]$$
$$\boldsymbol{run}_{ps}\ m = x \quad \text{where} \quad (\Omega, P, x) = m\ \left(\{()\}, \lambda\omega.1\right)$$

Fig. 4. State monad functions that represent queries and statements. The state is probability-space-valued.

If $K_L = \mathcal{R}[\![\mathsf{P_L}]\!]$ and $K_E = \mathcal{R}[\![\left(\mathsf{P_{E|L}}\ \mathsf{L}\right)]\!]$—which interpret (14)'s statements' right-hand sides—then $P = \mathcal{R}[\![(K_L\ L) \times (K_E\ E)]\!]$. This can be generalized.

Definition 1 (discrete product model). *Given a well-formed, discrete conditional theory $X_1 \sim e_1; \ldots; X_n \sim e_n$, let $K_i : \Omega \Rightarrow S_i \to [0,1]$, $K_i = \mathcal{R}[\![e_i]\!]$ for each $1 \le i \le n$. The* ***discrete product model*** *of the theory is*

$$\Omega = \bigtimes_{i=1}^{n} S_i \qquad X_i\ \omega = \omega_i\ (1 \le i \le n) \qquad P = \mathcal{R}\left[\!\!\left[\prod_{i=1}^{n} (K_i\ X_i)\right]\!\!\right] \tag{17}$$

Theorem 1 (semantic intent). *The discrete product model induces the stated conditional distributions and meets implicit independence assumptions.*

When writing distribution statements, practitioners tend to apply first-order distributions to simple random variables. But the discrete product model allows any λ_{ZFC} term e_i whose interpretation is a discrete **transition kernel** $\mathcal{R}[\![e_i]\!] : \Omega \Rightarrow S_i \to [0,1]$. In measure theory, transition kernels are used to build **product spaces** such as (Ω, P). Thus, $\mathcal{R}[\![\cdot]\!]$ links Bayesian practice to measure theory and represents an increase in expressive power in specifying distributions, by turning properly typed λ_{ZFC} terms into precisely what measure theory requires.

5.2 Interpreting Statements as Monadic Computations

Some conditional theories state more than just distributions [12, 21]. Interpreting theories with different kinds of statements requires recursive, rather than whole-theory, interpretation. Fortunately, well-formedness amounts to lexical scope, making it straightforward to interpret statements as monadic computations. We use the state monad with probability-space-valued state.

We assume the state monad's $\boldsymbol{return}_s$ and $\boldsymbol{bind}_s$. Fig. 4 shows the additional $\boldsymbol{dist}_{ps}$, $\boldsymbol{cond}_{ps}$ and $\boldsymbol{extend}_{ps}$. The first two simply reimplement $\boldsymbol{dist}$ and $\boldsymbol{cond}$. But $\boldsymbol{extend}_{ps}$, which interprets statements, needs more explanation.

$$\mathcal{M}[\![X_i := e_i;\ t_{i+1}]\!] = \boldsymbol{bind}_s\ (\boldsymbol{return}_s\ \mathcal{R}[\![e_i]\!])\ \lambda X_i.\mathcal{M}[\![t_{i+1}]\!]$$
$$\mathcal{M}[\![X_i \sim e_i;\ t_{i+1}]\!] = \boldsymbol{bind}_s\ (\boldsymbol{extend}_{ps}\ \mathcal{R}[\![e_i]\!])\ \lambda X_i.\mathcal{M}[\![t_{i+1}]\!]$$
$$\mathcal{M}[\![e_a = e_b;\ t_{i+1}]\!] = \boldsymbol{bind}_s\ (\boldsymbol{cond}_{ps}\ \mathcal{R}[\![e_a]\!]\ \mathcal{R}[\![e_b]\!])\ \lambda\_.\mathcal{M}[\![t_{i+1}]\!]$$
$$\mathcal{M}[\![\epsilon]\!] = \boldsymbol{return}_s\ (X_1, \ldots, X_n)$$

$$\mathbf{D}[\![e]\!]\ m = \boldsymbol{run}_{ps}\ (\boldsymbol{bind}_s\ m\ \lambda(X_1, \ldots, X_n).\boldsymbol{dist}_{ps}\ \mathcal{R}[\![e]\!])$$
$$\mathbf{D}[\![e_X \mid e_Y]\!]\ m = \lambda y.\mathbf{D}[\![e_X]\!]\ (\boldsymbol{bind}_s\ m\ \lambda(X_1, \ldots, X_n).\mathcal{M}[\![e_Y = y]\!])$$
$$\mathbf{P}[\![e]\!]\ m = \mathbf{D}[\![e]\!]\ m\ \mathit{true}, \quad \mathbf{P}[\![e_A \mid e_B]\!]\ m = \mathbf{D}[\![e_A \mid e_B]\!]\ m\ \mathit{true}\ \mathit{true}$$

Fig. 5. The conditional theory and query semantic functions.

According to (17), interpreting $X_i \sim e_i$ results in $\Omega_i = \Omega_{i-1} \times S_i$, with S_i extracted from $K_i : \Omega_{i-1} \Rightarrow S_i \to [0,1]$. A more precise type for K_i is the dependent type $(\omega : \Omega_{i-1}) \Rightarrow (S'_i\ \omega) \to [0,1]$, which reveals a complication. To extract S_i, we first must extract the random variable $S'_i : \Omega_{i-1} \to \mathcal{P}(S_i)$. So let $S'_i\ \omega = \boldsymbol{domain}\ (K_i\ \omega)$; then $S_i = \bigcup(\boldsymbol{image}\ S'_i\ \Omega_{i-1})$.

But this makes query implementation inefficient: if the union has little overlap or is disjoint, P will assign 0 to most ω. In more general terms, we actually have a *dependent* cartesian product $(\omega \in \Omega_{i-1}) \times (S'_i\ \omega)$, a generalization of the cartesian product.[3] To extend Ω, $\boldsymbol{extend}_{ps}$ calculates this product instead.

Dependent cartesian products are elegantly expressed using the set monad:

$$\boldsymbol{return}_v\ x = \{x\} \qquad \boldsymbol{bind}_v\ m\ f = \bigcup(\boldsymbol{image}\ f\ m) \tag{18}$$

Then $(a \in A) \times (B\ a) = \boldsymbol{bind}_v\ A\ \lambda a.\boldsymbol{bind}_v\ (B\ a)\ \lambda b.\boldsymbol{return}_v\ (a, b)$.

Fig. 5 defines $\mathcal{M}[\![\cdot]\!]$, which interprets conditional theories containing definition, distribution, and conditioning statements as probability space monad computations. After it exhausts the statements, it returns the random variables. Returning their names as well would be an obfuscating complication, which we avoid by implicitly extracting them from the theory before interpretation. (However, the implementation explicitly extracts and returns names.)

$\mathbf{D}[\![e]\!]$ expands to a distribution-valued computation and runs it with the ***empty probability space*** $(\Omega_0, P_0) = \big(\{()\}, \lambda\omega.1\big)$. $\mathbf{D}[\![e_X \mid e_Y]\!]$ conditions the probability space and hands off to $\mathbf{D}[\![e_X]\!]$. $\mathbf{P}[\![\cdot]\!]$ is defined in terms of $\mathbf{D}[\![\cdot]\!]$.

5.3 Approximating Models and Queries

We compute dependent cartesian products of sets represented by lazy lists in a way similar to enumerating $\mathbb{N} \times \mathbb{N}$. (It cannot be done with a monad as in the exact semantics, but we do not need it to.) The approximating versions of $\boldsymbol{dist}_{ps}$ and $\boldsymbol{cond}_{ps}$ apply ***finitize*** to the probability space.

[3] The dependent cartesian product also generalizes disjoint union to arbitrary index sets. It is often called a *dependent sum* and denoted $\Sigma a : A.(B\ a)$.

5.4 Implementation in Racket

$\mathcal{M}[\![\cdot]\!]$'s implementation is `MDL`. Like `RV`, it passes random variable identifiers, but it accumulates them. For example, `(MDL [] ([X ~ Px]))` expands to

```
([X] (bind/s (extend/ps (RV [] Px)) (λ (X) (ret/s (list X)))))
```

where `[X]` is the updated list of identifiers and the rest is a model computation.

We store theories in transformer bindings so queries can expand them later. For example, `(define-model die-roll [L ~ Pl] [E ~ (Pe/l L)])` expands to

```
(define-syntax die-roll #'(MDL [] ([L ~ Pl] [E ~ (Pe/l L)])))
```

The macro `with-model` introduces a scope in which a theory's variables are visible. For example, `(with-model die-roll (Dist L E))` looks up `die-roll` and expands it into its identifiers and computation. Using the identifiers as lambda arguments, `Dist` (the implementation of $\mathbf{D}[\![\cdot]\!]$) builds a query computation as in Fig. 5, and runs it with `(mapping (list empty) (λ (ω) 1))`, the empty probability space.

Using these identifiers would break hygiene, except that `Dist` replaces the lambda arguments' lexical context. This puts the theory's exported identifiers in scope, even when the theory and query are defined in separate modules. Because queries can access only the exported identifiers, it is safe.

Aside from passing identifiers and monkeying with hygiene, the macros are almost transcribed from the semantic functions.

Examples. Consider a conditional distribution with the first-order definition

```
(define (Geometric p)
  (mapping N1 (λ (n) (* p (expt (- 1 p) (- n 1))))))
```

where `N1` is a lazy list of natural numbers starting at `1`. Nahin gives a delightfully morbid use for `Geometric` in his book of probability puzzlers [15].

Two idiots duel with one gun. They put only one bullet in it, and take turns spinning the chamber and firing at each other. They know that if they each take one shot at a time, player one usually wins. Therefore, player one takes one shot, and after that, the next player takes one more shot than the previous player, spinning the chamber before each shot. How probable is player two's demise?

The distribution over the number of shots when the gun fires is `(Geometric 1/6)`. Using this procedure to determine whether player one fires shot `n`:

```
(define (p1-fires? n [shots 1])
  (cond [(n . <= . 0)  #f]
        [else  (not (p1-fires? (- n shots) (add1 shots)))]))
```

we compute the probability that player one wins with

```
(with-model (model [winning-shot ~ (Geometric 1/6)])
  (Pr (p1-fires? winning-shot)))
```

Nahin computes 0.523919127555099524791984 3—25 decimal digits—with custom MATLAB code. At `appx-z` $\geq$ `321`, our solution computes the same digits. (Though it appends the digits 9..., so Nahin should have rounded up!) Implementing it took about five minutes. But the problem is not Bayesian.

This is: suppose player one slyly suggests a single coin flip to determine whether they spin the chamber before each shot. You do not see the duel, but learn that player two won. What is the probability they spun the chamber?

Suppose that the well-known `Bernoulli` and discrete `Uniform` conditional distributions are defined. Using these first-order conditional distributions and Racket's `cond`, we can state a fairly direct theory of the duel:

```
(define-model half-idiot-duel
  [spin? ~ (Bernoulli 1/2)]
  [winning-shot ~ (cond [spin?  (Geometric 1/6)]
                        [else   (Uniform 1 6)])])
```

Then `(Pr spin? (not (p1-fires? winning-shot)))` converges to about `0.588`.

Bayesian practitioners would normally create a new first-order conditional distribution `WinningShot`, and then state `[winning-shot ~ (WinningShot spin?)]`. Most would *like* to state something more direct—such as the above theory, which plainly shows how `spin?`'s value affects `winning-shot`'s distribution. However, without a semantics, they cannot be sure that using the value of a `cond` (or of any "if"-like expression) as a distribution is well-defined. That `winning-shot` has a *different range* for each value of `spin?` makes things more uncertain.

As specified by $\mathcal{R}[\![\cdot]\!]$, our implementation interprets `(cond ...)` above as a stochastic transition kernel. As specified by $\mathcal{M}[\![\cdot]\!]$, it builds the probability space using dependent cartesian products. Thus, the direct theory really is well-defined.

The most direct theory has infinitely many statements, one for each possible shot. Supporting such theories is future work.

6 Why Separate Statements and Queries?

Whether queries should be allowed inside theories is a decision with subtle effects.

Theories are sets of facts. Well-formedness imposes a partial order, but every linearization should be interpreted equivalently. Thus, we can determine whether two kinds of statements can coexist in theories by determining whether they can be exchanged without changing the interpretation. This is equivalent to determining whether the corresponding monad functions commute.

The following definitions suppose a conditional theory $t_1; \ldots; t_n$ in which exchanging some t_i and t_{i+1} (where $i < n$) is well-formed. Applying semantic functions in the definitions yields definitions that are independent of syntax but difficult to read, so we give the syntactic versions.

Definition 2 (commutativity). *We say that t_i and t_{i+1}* ***commute*** *when* $\mathcal{M}[\![t_1; \ldots; t_i; t_{i+1}; \ldots; t_n]\!]\ (\Omega_0, P_0) = \mathcal{M}[\![t_1; \ldots; t_{i+1}; t_i; \ldots; t_n]\!]\ (\Omega_0, P_0)$.

This notion of commutativity is too strong: distribution statements would never commute with each other. We need a weaker test than equality.

Definition 3 (equivalence in distribution). *Suppose $X_1, \ldots, X_k$ are defined in $t_1, \ldots, t_n$. Let $m = \mathcal{M}[\![t_1, \ldots, t_n]\!]$, and m' be a (usually different) probability space monad computation. We write $m \equiv_{\mathbf{D}} m'$ and call m and m'* ***equivalent in distribution*** *when* $\mathbf{D}[\![X_1, \ldots, X_k]\!]\ m = \mathbf{D}[\![X_1, \ldots, X_k]\!]\ m'$.

The following says $\equiv_{\mathbf{D}}$ is like observational equivalence with query contexts:

Theorem 2 (context). $\mathbf{D}[\![e_X \mid e_Y]\!]\ m = \mathbf{D}[\![e_X \mid e_Y]\!]\ m'$ *for all random variables* $\mathcal{R}[\![e_X]\!]$ *and* $\mathcal{R}[\![e_Y]\!]$ *if and only if* $m \equiv_{\mathbf{D}} m'$.

Definition 4 (commutativity in distribution). *We say* t_i *and* t_{i+1} *commute* ***in distribution*** *when* $\mathcal{M}[\![t_1; \ldots; t_i; t_{i+1}; \ldots; t_n]\!] \equiv_{\mathbf{D}} \mathcal{M}[\![t_1; \ldots; t_{i+1}; t_i; \ldots; t_n]\!]$.

Theorem 3. *The following table summarizes commutativity of* $\boldsymbol{cond}_{ps}$, $\boldsymbol{dist}_{ps}$ *and* $\boldsymbol{extend}_{ps}$ *in the probability space monad:*

$\boldsymbol{cond}_{ps}$	$=$		
$\boldsymbol{extend}_{ps}$	$=$	$\equiv_{\mathbf{D}}$	
$\boldsymbol{dist}_{ps}$	$\not\equiv_{\mathbf{D}}$	$=$	$=$
	$\boldsymbol{cond}_{ps}$	$\boldsymbol{extend}_{ps}$	$\boldsymbol{dist}_{ps}$

By Thm. 3, if we are to maintain the idea that theories are sets of facts, we cannot allow both conditioning and query statements.

7 Related Work

Our approach to semantics is similar to abstract interpretation: we have a concrete (exact) semantics and a family of abstractions parameterized by z (approximating semantics). We have not framed our approach this way because our approximations are not conservative, and would be difficult to formulate as abstractions when parameterized on a random source (which we intend to do).

Bayesian practitioners occasionally create languages for modeling and queries. Analyzing their properties is usually difficult, as they tend to be defined by implementations. Almost all of them compute converging approximations and support conditional queries. When they work as expected, they are useful.

Koller and Pfeffer [9] efficiently compute exact distributions for the outputs of programs in a Scheme-like language. BUGS [11] focuses on efficient approximate computation for probabilistic theories with a finitely many statements, with distributions that practitioners typically use. BLOG [14] exists specifically to allow stating distributions over countably infinite vectors. BLAISE [1] allows stating both distribution and approximation method for each random variable. Church [4] is a Scheme-like probabilistic language with approximate inference, and focuses on expressiveness.

Kiselyov [8] embeds a probabilistic language in O'Caml for efficient computation. It uses continuations to enumerate or sample random variable values, and has a `fail` construct for the *complement* of conditioning. The sampler looks ahead for `fail` and can handle it efficiently. This may be justified by commutativity (Thm. 3), depending on interaction with other language features.

There is a fair amount of semantics work in probabilistic languages. Most of it is not motivated by Bayesian concerns, and thus does not define conditioning. Kozen [10] defines the meaning of bounded-space, imperative "while" programs as functions from probability measures to probability measures. Hurd [6] proves

properties about programs with binary random choice by encoding programs and portions of measure theory in HOL.

Jones [7] develops a domain-theoretic variation of probability theory, and with it defines the probability monad, whose discrete version is a distribution-valued variation of the set or list monad. Ramsey and Pfeffer [20] define the probability monad measure-theoretically and implement a language for finite probability. We do not build on this work because the probability monad does not build a probability space, making it difficult to reason about conditioning.

Pfeffer also develops IBAL [19], apparently the only lambda calculus with finite probabilistic choice that also defines conditional queries. Park [16] extends a lambda calculus with probabilistic choice, defining it for a very general class of probability measures using inverse transform sampling.

8 Conclusions and Future Work

For discrete Bayesian theories, we explained a large subclass of notation as measure-theoretic calculations by transformation into λ_{ZFC}. There is now at least one precisely defined set of expressions that denote discrete conditional distributions in conditional theories, and it is very large and expressive. We gave a converging approximating semantics and implemented it in Racket.

Now that we are satisfied that our approach works, we turn our attention to uncountable sample spaces and theories with infinitely many statements.

Following measure-theoretic structure in our preliminary work should make the transition to uncountable spaces fairly smooth. The functional structure of the exact semantics will not change, but some details will. The random variable idiom will be identical, but will require measurability proofs. We will still interpret statements as state monad computations, but with general probability spaces as state instead of discrete probability spaces. We will use regular conditional probability in $\boldsymbol{cond}_{ps}$, $\boldsymbol{extend}_{ps}$ will calculate product σ-algebras and transition kernel products, and $\boldsymbol{dist}_{ps}$ will return probability measures. We will not need to change $\mathcal{R}[\![\cdot]\!]$, $\mathbf{D}[\![\cdot]\!]$ or $\mathbf{P}[\![\cdot]\!]$. Many approximations are available; the most efficient and general are sampling methods. We will likely choose sampling methods that parallelize easily.

The most general constructive way to specify theories with infinitely many primitive random variables is with recursive abstractions, but it is not clear what kind of abstraction we need. Lambdas are suitable for most functional programming, in which it is usually good that intermediate values are unobservable. However, they do not meet Bayesian needs: practitioners define theories to study them, not to obtain single answers. If lambdas were the only abstraction, returning every intermediate value from every lambda would become *good practice*. Because we do not know what form abstraction will take, we will likely develop it independently by allowing theories with infinitely many statements.

Model equivalence in distribution extends readily to uncountable spaces. It defines a standard for measure-theoretic optimizations, which can only be done in the exact semantics. Examples are variable collapse, a probabilistic analogue

of constant folding that can increase efficiency by an order of magnitude, and a probabilistic analogue of constraint propagation to speed up conditional queries.

References

1. Bonawitz, K.A.: Composable Probabilistic Inference with Blaise. Ph.D. thesis, Massachusetts Institute of Technology (2008)
2. Culpepper, R.: Refining Syntactic Sugar: Tools for Supporting Macro Development. Ph.D. thesis, Northeastern University (2010), to Appear
3. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc. (2010), http://racket-lang.org/tr1/
4. Goodman, N., Mansinghka, V., Roy, D., Bonawitz, K., Tenenbaum, J.: Church: a language for generative models. In: Uncertainty in Artificial Intelligence (2008)
5. Gordon, M.: Higher order logic, set theory or both? (1996), invited talk, TPHOLs, Turku, Finland
6. Hurd, J.: Formal Verification of Probabilistic Algorithms. Ph.D. thesis, University of Cambridge (2002)
7. Jones, C.: Probabilistic Non-Determinism. Ph.D. thesis, University of Edinburgh (1990)
8. Kiselyov, O., Shan, C.: Monolingual probabilistic programming using generalized coroutines. In: Uncertainty in Artificial Intelligence (2008)
9. Koller, D., McAllester, D., Pfeffer, A.: Effective Bayesian inference for stochastic programs. In: 14th National Conference on Artificial Intelligence (August 1997)
10. Kozen, D.: Semantics of probabilistic programs. In: Foundations of Computer Science (1979)
11. Lunn, D.J., Thomas, A., Best, N., Spiegelhalter, D.: WinBUGS – a Bayesian modelling framework. Statistics and Computing 10(4) (2000)
12. Mateescu, R., Dechter, R.: Mixed deterministic and probabilistic networks. Annals of Mathematics and Artificial Intelligence (2008)
13. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(1) (2008)
14. Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D., Kolobov, A.: BLOG: Probabilistic models with unknown objects. In: International Joint Conference on Artificial Intelligence (2005)
15. Nahin, P.J.: Duelling Idiots and Other Probability Puzzlers. Princeton University Press (2000)
16. Park, S., Pfenning, F., Thrun, S.: A probabilistic language based upon sampling functions. Transactions on Programming Languages and Systems 31(1) (2008)
17. Paulson, L.C.: Set theory for verification: I. From foundations to functions. Journal of Automated Reasoning 11, 353–389 (1993)
18. Paulson, L.C.: Set theory for verification: II. Induction and recursion. Journal of Automated Reasoning 15, 167–215 (1995)
19. Pfeffer, A.: The design and implementation of IBAL: A general-purpose probabilistic language. In: Statistical Relational Learning. MIT Press (2007)
20. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: Principles of Programming Languages (2002)
21. Toronto, N., Morse, B.S., Seppi, K., Ventura, D.: Super-resolution via recapture and Bayesian effect modeling. In: Computer Vision and Pattern Recognition (2009)