From Bayesian Notation to Pure Racket*

via Discrete Measure-Theoretic Probability in λ_{ZFC}

Implementation and Application of Functional Languages

September 1-3, 2010

Neil Toronto and Jay McCarthy

PLT @ Brigham Young University, Utah, USA

* Formerly PLT-Scheme

I'm Neil Toronto of Toronto and McCarthy, from the PLT group at Brigham Young University. We're developing languages for Bayesian modeling and queries.

Bayesian Practice and Philosophy

- State a probabilistic **model** of a process; then pose a **query** that runs the process backwards

Doing Bayesian statistics is like doing physics, but for fuzzy, uncertain, or random things. In physics, you might use a Newtonian model that gives distance in terms of time (i.e. d=rt) and ask how much time it will take to travel given a certain distance.

Bayesian Practice and Philosophy

- State a probabilistic **model** of a process; then pose a **query** that runs the process backwards
  - Document generation / Is this email spam?

In Bayesian statistics, you might design a model that says which words tend to appear in a document given the writer's intent, and ask how likely it is that the writer is trying to sell you Viagra given the words in an email.

Bayesian Practice and Philosophy

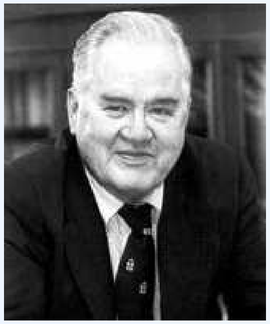
- State a probabilistic **model** of a process; then pose a **query** that runs the process backwards
  - Document generation / Is this email spam?
  - Real-world scenes and image capture / Likely scene given a photograph

Or you could model real-world scenes and image capture, and then ask for a likely scene given an image---and that's the computer vision problem in Bayesian terms.

Bayesian Practice and Philosophy

- State a probabilistic **model** of a process; then pose a **query** that runs the process backwards
  - Document generation / Is this email spam?
  - Real-world scenes and image capture / Likely scene given a photograph

---



John Wilder Tukey

"An approximate answer to the right question is worth a great deal more than a precise answer to the wrong question."

Bayesian statisticians follow a particular philosophy when they model the world and ask questions about it. John Tukey probably stated it best: (read).

Philosophy Into Practice (1)

- Approximations must be put off as long as possible

In practice, the philosophy requires putting off approximation for as long as possible. In fact, Bayesians intentionally forget about how hard it might be to calculate answers when they design models and pose queries.

Philosophy Into Practice (1)

- Approximations must be put off as long as possible
  - Models and queries are exact, and generally not closed-form nor finitely computable

The answers end up not being closed-form or finitely computable...

Philosophy Into Practice (1)

- Approximations must be put off as long as possible
  - Models and queries are exact, and generally not closed-form nor finitely computable
  - Compute answers as converging approximations

... so they usually end up compiling their queries by hand, into programs that compute converging approximations.

Philosophy Into Practice (1)

- Approximations must be put off as long as possible
  - Models and queries are exact, and generally not closed-form nor finitely computable
  - Compute answers as converging approximations
- Example: enlarging images

Model and query

$$\mathbf{S}^{\theta}_{i,j} \sim \text{Uniform}(-\pi, \pi) \qquad \mathbf{S}^{v^+}_{i,j} \sim \text{Uniform}(0, 1)$$
$$\mathbf{S}^{d}_{i,j} \sim \text{Uniform}(-3, 3) \qquad \mathbf{S}^{v^-}_{i,j} \sim \text{Uniform}(0, 1)$$
$$\mathbf{S}^{\sigma}_{i,j} \sim \text{Beta}(1.6, 1)$$

$$\mathbf{I}_{i,j} | \mathbf{S}_{\mathrm{N9}(i,j)} \sim \text{Normal}(\mathbf{E}[\mathbf{S}_{i,j}], \omega)$$
$$\Phi_{i,j}(\mathbf{S}_{\mathrm{N9}(i,j)}) \equiv \exp\left(-\frac{\text{Var}[\mathbf{S}_{i,j}]}{2\gamma^2}\right)$$

What is the distribution of $\mathbf{I}'|\mathbf{I}$?

Here's a concrete example: an idealized model of taking pictures of real-world scenes. The query asks for a likely scene given an image, and then for a higher-resolution picture of the same scene. In other words, it enlarges images. The model and query are short and elegant.

Philosophy Into Practice (1)

- Approximations must be put off as long as possible
  - Models and queries are exact, and generally not closed-form nor finitely computable
  - Compute answers as converging approximations
- Example: enlarging images

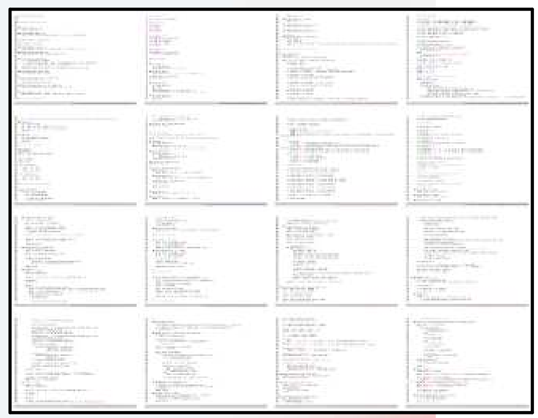
Model and query

$$\mathbf{S}^{\theta}_{i,j} \sim \mathrm{Uniform}(-\pi, \pi) \qquad \mathbf{S}^{v+}_{i,j} \sim \mathrm{Uniform}(0,1)$$
$$\mathbf{S}^{d}_{i,j} \sim \mathrm{Uniform}(-3, 3) \qquad \mathbf{S}^{v-}_{i,j} \sim \mathrm{Uniform}(0,1)$$
$$\mathbf{S}^{\sigma}_{i,j} \sim \mathrm{Beta}(1.6, 1)$$

$$\mathbf{I}_{i,j} | \mathbf{S}_{N9(i,j)} \sim \mathrm{Normal}(\mathbf{E}[\mathbf{S}_{i,j}], \omega)$$
$$\Phi_{i,j}(\mathbf{S}_{N9(i,j)}) \equiv \exp\left(-\frac{\mathrm{Var}[\mathbf{S}_{i,j}]}{2\gamma^2}\right)$$

What is the distribution of $\mathbf{I}'|\mathbf{I}$?

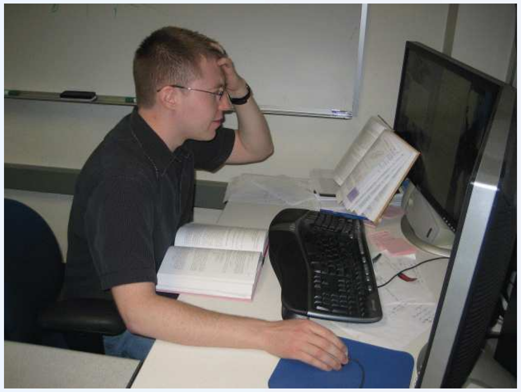
The answer's approximation



And this is the 600 lines of hand-compiled, highly vectorized Python code that computes an approximate answer.

Philosophy Into Practice (2)

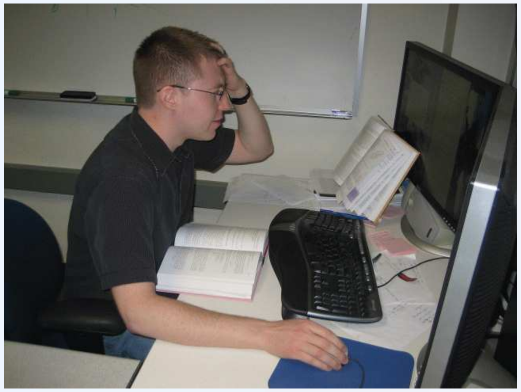
- Grads become compilers (and are just as grumpy)



These are my friends, Andrew and Dave, who write programs like that, in their natural habitat, under typical conditions. They suffer these conditions because above all, they want to answer the right questions.

Philosophy Into Practice (2)

- Grads become compilers (and are just as grumpy)

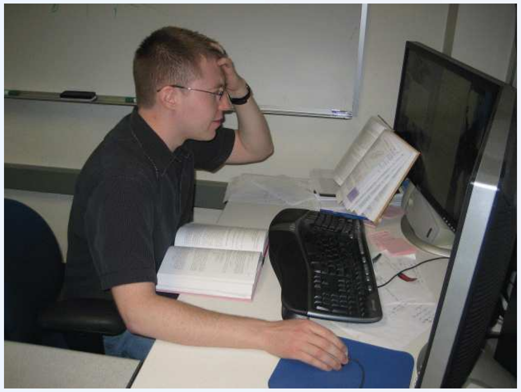


- Our motivation: free (fire?) the grad students

So Bayesian philosophy motivates our work: we want to free Andrew and Dave from compiling answers to queries by hand. If we did that, they could spend a lot more time searching the space of models and experimenting.

Philosophy Into Practice (2)

- Grads become compilers (and are just as grumpy)



- Our motivation: free (fire?) the grad students
- Our primary constraints
  - Do not approximate earlier than users would
  - Do not force users to approximate early

The philosophy also places two constraints on our work: our query implementations can't approximate early, and the modeling language has to be expressive enough that it doesn't force users to approximate early. Otherwise, they won't be our users.

Philosophical Constraints

- Compatible approach
  1. Informally determine meaning of notation

Here's an approach that satisfies the constraints. First, we determine what the notation means, which usually means turning examples into something formal and compositional enough to generalize.

Philosophical Constraints

- Compatible approach
  1. Informally determine meaning of notation
  2. Develop exact $[\![\cdot]\!]$: “notation” → “calculations”

Then we develop an exact, compositional semantics...

Philosophical Constraints

- Compatible approach
  1. Informally determine meaning of notation
  2. Develop exact $[\![\cdot]\!]$: "notation" → "calculations"
  3. Approximate $[\![\cdot]\!]$, prove convergence

... approximate the calculations, prove that the approximation converges, and then...

Philosophical Constraints

- Compatible approach
  1. Informally determine meaning of notation
  2. Develop exact $[\![\cdot]\!]$: "notation" → "calculations"
  3. Approximate $[\![\cdot]\!]$, prove convergence
  4. Implement approximating $[\![\cdot]\!]$

... implement the approximating semantics in Racket.

Philosophical Constraints

- Compatible approach
  1. Informally determine meaning of notation
  2. Develop exact $[\![\cdot]\!]$: “notation” → “calculations”
  3. Approximate $[\![\cdot]\!]$, prove convergence
  4. Implement approximating $[\![\cdot]\!]$
- Analogous to abstract interpretation

concrete/exact, abstract/approximating

If you know abstract interpretation, our approach should seem very familiar. It isn't abstract interpretation, though, because the approximations aren't conservative.

Which Probability Theory?

- Naive/undergraduate/informal probability theory
  - How Bayesians tend to think about probability

To turn notation into exact calculations, we need a theory of probability that tells us what those calculations should be. Bayesians tend to think and calculate using naive probability, which you probably learned if you had to take an undergraduate statistics course. But we can't use naive probability.

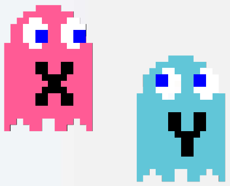
Which Probability Theory?

- Naive/undergraduate/informal probability theory
  - How Bayesians tend to think about probability
  - But can’t properly express infinities

We plan to allow infinitely many random variables and distributions that have both discrete and continuous parts. Bayesians want those things, but naive probability can't explain them.

Which Probability Theory?

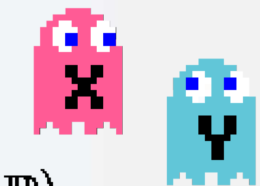
- Naive/undergraduate/informal probability theory
  - How Bayesians tend to think about probability
  - But can’t properly express infinities
  - “Spooky interaction at a distance”



The second problem arises from the fact that random variables can interact non-locally, similar to how variables in languages with mutable state can interact non-locally. Naive probability theory doesn't explain this compositionally.

Which Probability Theory?

- Naive/undergraduate/informal probability theory
  - How Bayesians tend to think about probability
  - But can’t properly express infinities
  - “Spooky interaction at a distance”

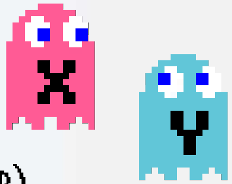


- Measure-theoretic probability: global $(\Omega, \Sigma, \mathbb{P})$

Measure-theoretic probability explains non-local interaction by having all random variables interact through a single, global object. You might call it a `store,' but its actual name is `probability space.'

Which Probability Theory?

- Naive/undergraduate/informal probability theory
  - How Bayesians tend to think about probability
  - But can’t properly express infinities
  - “Spooky interaction at a distance”

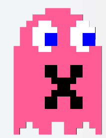


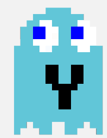
- Measure-theoretic probability: global $(\Omega, \Sigma, \mathbb{P})$
  - Ω : set: all possible “worlds”

Omega is the set of all possible states of the world being modeled. For example, if your model includes flipping two coins, all four combinations of outcomes will be encoded somehow in Omega.

Which Probability Theory?

- Naive/undergraduate/informal probability theory
  - How Bayesians tend to think about probability
  - But can't properly express infinities
  - "Spooky interaction at a distance"



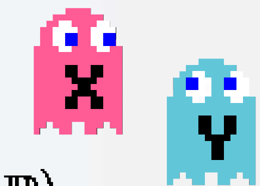


- Measure-theoretic probability: global $(\Omega, \Sigma, \mathbb{P})$
  - $\Omega : \mathtt{set}$: all possible "worlds"
  - $X : \Omega \to S$: "getters" or "readers" for worlds

You'll also have random variables that represent the coin flips, which decode or `read' the outcomes from a world object.

Which Probability Theory?

- Naive/undergraduate/informal probability theory
  - How Bayesians tend to think about probability
  - But can’t properly express infinities
  - “Spooky interaction at a distance”

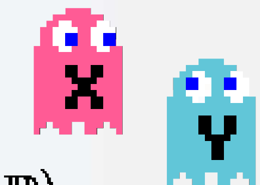


- Measure-theoretic probability: global $(\Omega, \Sigma, \mathbb{P})$
  - Ω : set: all possible “worlds”
  - $X : \Omega \to S$: “getters” or “readers” for worlds
  - Σ : set: measurable events (for uncountable Ω)

Sigma is critical for handling uncountable infinities properly, and it's complicated.

Which Probability Theory?

- Naive/undergraduate/informal probability theory
  - How Bayesians tend to think about probability
  - But can't properly express infinities
  - "Spooky interaction at a distance"

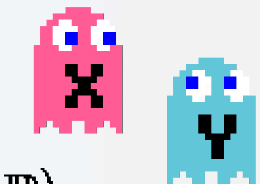


- Measure-theoretic probability: global $(\Omega, \Sigma, \mathbb{P})$
  - $\Omega : \mathbf{set}$: all possible "worlds"
  - $X : \Omega \to S$: "getters" or "readers" for worlds
  - $\Sigma : \mathbf{set}$: measurable events (for uncountable Ω)
  - $\mathbb{P} : \Sigma \to [0, 1]$ (or $P : \Omega \to [0, 1]$): probabilities of events

And last, P assigns probabilities. In our preliminary work, we don't let Omega get any bigger than countable, so we can forget about Sigma, and use a P that assigns probabilities to single worlds. We have simpler calculations that way, but they're still structured measure-theoretically, and we still have to deal with approximation.

Which Probability Theory?

- Naive/undergraduate/informal probability theory
  - How Bayesians tend to think about probability
  - But can't properly express infinities
  - "Spooky interaction at a distance"



- Measure-theoretic probability: global $(\Omega, \Sigma, \mathbb{P})$
  - Ω : $\mathbf{set}$: all possible "worlds"
  - $X : \Omega \rightarrow S$: "getters" or "readers" for worlds
  - Σ : $\mathbf{set}$: measurable events (for uncountable Ω)
  - $\mathbb{P} : \Sigma \rightarrow [0,1]$ (or $P : \Omega \rightarrow [0,1]$): probabilities of events
  - Calculations uncomputable when Ω infinite

Measure theory fits naturally into functional programming, but before we start defining monads, we have to deal with the fact that measure-theoretic calculations aren't generally computable.

Lambda-ZFC

- Looking for language for compositional measure- theoretic calculations; similarity to Racket a plus

We want to transform notation into measure-theoretic calculations, and eventually approximate the calculations in Racket. For the semantic function's target language, then, we need a call-by-value lambda calculus for expressing uncomputable things that are well-defined in contemporary mathematics.

Lambda-ZFC

- Looking for language for compositional measure- theoretic calculations; similarity to Racket a plus

λ calculus

$((\lambda x.e, x, e\ e), \alpha, \beta)$

So we start with Alonzo Church's invention, the lambda calculus; then we add Ernst Zermelo and Abraham Fraenkel's inventions, the well-founded sets and set operations; and we get lambda-ZFC.

Lambda-ZFC

- Looking for language for compositional measure- theoretic calculations; similarity to Racket a plus

+

λ calculus
$((\lambda x.e, x, e\ e), \alpha, \beta)$

Set theory
$(V, \in, =, \{\cdot\}, \bigcup, image, \mathcal{P}, order)$

So we start with Alonzo Church's invention, the lambda calculus; then we add Ernst Zermelo and Abraham Fraenkel's inventions, the well-founded sets and set operations; and we get lambda-ZFC.

Lambda-ZFC

- Looking for language for compositional measure- theoretic calculations; similarity to Racket a plus

+

=

λ calculus
$((\lambda x.e, x, e\ e), \alpha, \beta)$

Set theory
$(V, \in, =, \{\cdot\}, \bigcup, \mathit{image}, \mathcal{P}, \mathit{order})$

λ_{ZFC}

So we start with Alonzo Church's invention, the lambda calculus; then we add Ernst Zermelo and Abraham Fraenkel's inventions, the well-founded sets and set operations; and we get lambda-ZFC.

Lambda-ZFC

- Looking for language for compositional measure- theoretic calculations; similarity to Racket a plus

+

=

λ calculus

$((\lambda x.e, x, e\ e), \alpha, \beta)$

Set theory

$(V, \in, =, \{\cdot\}, \bigcup, \mathit{image}, \mathcal{P}, \mathit{order})$

λ_{ZFC}

- “Programming” is doing contemporary math, plus λx.e

Programming in lambda-ZFC is like doing contemporary mathematics... but with first-class lambdas, so we can structure our uncomputable measure-theoretic calculations as monadic computations.

Lambda-ZFC

- Looking for language for compositional measure- theoretic calculations; similarity to Racket a plus

+

=

λ calculus
$((\lambda x.e, x, e\ e), \alpha, \beta)$

Set theory
$(V, \in, =, \{\cdot\}, \bigcup, \mathit{image}, \mathcal{P}, \mathit{order})$

λ_{ZFC}

- “Programming” is doing contemporary math, plus λx.e
- Contains all set-theoretic functions

Lambda-ZFC contains all set-theoretic functions; specifically, all conditional probability distributions.

Lambda-ZFC

- Looking for language for compositional measure- theoretic calculations; similarity to Racket a plus

λ calculus
$((\lambda x.e, x, e\ e), \alpha, \beta)$

Set theory
$(V, \in, =, \{\cdot\}, \bigcup, image, \mathcal{P}, order)$

λ_{ZFC}

- “Programming” is doing contemporary math, plus λx.e
- Contains all set-theoretic functions
- Can solve any OTM halting problem constructively

To give you an idea of its relative computational power: you can solve any oracle Turing machine halting problem by writing an interpreter in lambda-ZFC. It might seem like too much power, but remember that we want to interpret Bayesian notation exactly. We'll worry about computability when we do the approximations.

Interpreting Notation

Syntactic Category	Examples	Computational Structure	Semantic Functions

Developing the whole semantics right now would take too much time, so I'm going to give some examples of syntax and talk about the structure of the calculations.

First we have random variable expressions. In the first example, X and Y are random variables, so they're functions of Omega. I've already hinted that you could interpret this by regarding random variables as reader monad computations.

But there's no reason to impose a total order, so we use the corresponding applicative functor, or idiom.

Next, we have statements about random variables. A collection of statements is a probabilistic model. We interpret each statement as transforming the global probability space. The first example, X is distributed Geometric B, extends the probability space. The second example is a `condition,' which asserts that applying the random variable X+Y to any world must yield 4. It *restricts* the global probability space.

A nice way to structure these calculations is with the state monad, with probability-space-valued state.

Last, we have queries. The first example is a `conditional probability query'. It conditions the probability space first, and then asks for the probability that B outputs 1/2. The second example is like the first, but is parameterized on the outputs of B and X+Y. It should return a function, or a distribution, so it's a `distribution query'.

Queries run the probability space monad computation in their own particular way.

Interpreting Notation

Syntactic Category	Examples	Computational Structure	Semantic Functions
Expressions	$X + Y$ $\text{Geometric}(B)$		

Developing the whole semantics right now would take too much time, so I'm going to give some examples of syntax and talk about the structure of the calculations.

First we have random variable expressions. In the first example, X and Y are random variables, so they're functions of Omega. I've already hinted that you could interpret this by regarding random variables as reader monad computations.

But there's no reason to impose a total order, so we use the corresponding applicative functor, or idiom.

Next, we have statements about random variables. A collection of statements is a probabilistic model. We interpret each statement as transforming the global probability space. The first example, X is distributed Geometric B, extends the probability space. The second example is a `condition,' which asserts that applying the random variable X+Y to any world must yield 4. It *restricts* the global probability space.

A nice way to structure these calculations is with the state monad, with probability-space-valued state.

Last, we have queries. The first example is a `conditional probability query'. It conditions the probability space first, and then asks for the probability that B outputs 1/2. The second example is like the first, but is parameterized on the outputs of B and X+Y. It should return a function, or a distribution, so it's a `distribution query'.

Queries run the probability space monad computation in their own particular way.

Interpreting Notation

Syntactic Category	Examples	Computational Structure	Semantic Functions
Expressions	$X + Y$ $\mathrm{Geometric}(B)$	Environment idiom: $R\ a = \Omega \rightarrow a$	$\mathcal{R}[\![\cdot]\!]$, **RV**

Developing the whole semantics right now would take too much time, so I'm going to give some examples of syntax and talk about the structure of the calculations.

First we have random variable expressions. In the first example, X and Y are random variables, so they're functions of Omega. I've already hinted that you could interpret this by regarding random variables as reader monad computations.

But there's no reason to impose a total order, so we use the corresponding applicative functor, or idiom.

Next, we have statements about random variables. A collection of statements is a probabilistic model. We interpret each statement as transforming the global probability space. The first example, X is distributed Geometric B, extends the probability space. The second example is a `condition,' which asserts that applying the random variable X+Y to any world must yield 4. It *restricts* the global probability space.

A nice way to structure these calculations is with the state monad, with probability-space-valued state.

Last, we have queries. The first example is a `conditional probability query'. It conditions the probability space first, and then asks for the probability that B outputs 1/2. The second example is like the first, but is parameterized on the outputs of B and X+Y. It should return a function, or a distribution, so it's a `distribution query'.

Queries run the probability space monad computation in their own particular way.

Interpreting Notation

Syntactic Category	Examples	Computational Structure	Semantic Functions
Expressions	$X + Y$ $\mathrm{Geometric}(B)$	Environment idiom: $R\ a = \Omega \to a$	$\mathcal{R}[\![\cdot]\!]$, **RV**
Statements	$X \sim \mathrm{Geometric}(B)$ $X + Y = 4$		

Developing the whole semantics right now would take too much time, so I'm going to give some examples of syntax and talk about the structure of the calculations.

First we have random variable expressions. In the first example, X and Y are random variables, so they're functions of Omega. I've already hinted that you could interpret this by regarding random variables as reader monad computations.

But there's no reason to impose a total order, so we use the corresponding applicative functor, or idiom.

Next, we have statements about random variables. A collection of statements is a probabilistic model. We interpret each statement as transforming the global probability space. The first example, X is distributed Geometric B, extends the probability space. The second example is a `condition,' which asserts that applying the random variable X+Y to any world must yield 4. It *restricts* the global probability space.

A nice way to structure these calculations is with the state monad, with probability-space-valued state.

Last, we have queries. The first example is a `conditional probability query'. It conditions the probability space first, and then asks for the probability that B outputs 1/2. The second example is like the first, but is parameterized on the outputs of B and X+Y. It should return a function, or a distribution, so it's a `distribution query'.

Queries run the probability space monad computation in their own particular way.

Interpreting Notation

Syntactic Category	Examples	Computational Structure	Semantic Functions
Expressions	$X + Y$ $\text{Geometric}(B)$	Environment idiom: $R\ a = \Omega \to a$	$\mathcal{R}[\![\cdot]\!]$, **RV**
Statements	$X \sim \text{Geometric}(B)$ $X + Y = 4$	State monad: $M\ b = PS \to (PS, b)$ (usually $b = R\ a$)	$\mathcal{M}[\![\cdot]\!]$, **`model`**

Developing the whole semantics right now would take too much time, so I'm going to give some examples of syntax and talk about the structure of the calculations.

First we have random variable expressions. In the first example, X and Y are random variables, so they're functions of Omega. I've already hinted that you could interpret this by regarding random variables as reader monad computations.

But there's no reason to impose a total order, so we use the corresponding applicative functor, or idiom.

Next, we have statements about random variables. A collection of statements is a probabilistic model. We interpret each statement as transforming the global probability space. The first example, X is distributed Geometric B, extends the probability space. The second example is a `condition,' which asserts that applying the random variable X+Y to any world must yield 4. It *restricts* the global probability space.

A nice way to structure these calculations is with the state monad, with probability-space-valued state.

Last, we have queries. The first example is a `conditional probability query'. It conditions the probability space first, and then asks for the probability that B outputs 1/2. The second example is like the first, but is parameterized on the outputs of B and X+Y. It should return a function, or a distribution, so it's a `distribution query'.

Queries run the probability space monad computation in their own particular way.

Interpreting Notation

Syntactic Category	Examples	Computational Structure	Semantic Functions
Expressions	$X + Y$ $\mathrm{Geometric}(B)$	Environment idiom: $R\ a = \Omega \to a$	$\mathcal{R}[\![\cdot]\!]$, **RV**
Statements	$X \sim \mathrm{Geometric}(B)$ $X + Y = 4$	State monad: $M\ b = PS \to (PS, b)$ (usually $b = R\ a$)	$\mathcal{M}[\![\cdot]\!]$, **`model`**
Queries	$\mathbf{P}\left[B = \frac{1}{2} \mid X + Y = 4\right]$ $\mathcal{L}[B \mid X + Y]$		

Developing the whole semantics right now would take too much time, so I'm going to give some examples of syntax and talk about the structure of the calculations.

First we have random variable expressions. In the first example, X and Y are random variables, so they're functions of Omega. I've already hinted that you could interpret this by regarding random variables as reader monad computations.

But there's no reason to impose a total order, so we use the corresponding applicative functor, or idiom.

Next, we have statements about random variables. A collection of statements is a probabilistic model. We interpret each statement as transforming the global probability space. The first example, X is distributed Geometric B, extends the probability space. The second example is a `condition,' which asserts that applying the random variable X+Y to any world must yield 4. It *restricts* the global probability space.

A nice way to structure these calculations is with the state monad, with probability-space-valued state.

Last, we have queries. The first example is a `conditional probability query'. It conditions the probability space first, and then asks for the probability that B outputs 1/2. The second example is like the first, but is parameterized on the outputs of B and X+Y. It should return a function, or a distribution, so it's a `distribution query'.

Queries run the probability space monad computation in their own particular way.

Interpreting Notation

Syntactic Category	Examples	Computational Structure	Semantic Functions
Expressions	$X + Y$ $\mathrm{Geometric}(B)$	Environment idiom: $R\ a = \Omega \to a$	$\mathcal{R}[\![\cdot]\!]$, **`RV`**
Statements	$X \sim \mathrm{Geometric}(B)$ $X + Y = 4$	State monad: $M\ b = PS \to (PS, b)$ (usually $b = R\ a$)	$\mathcal{M}[\![\cdot]\!]$, **`model`**
Queries	$\mathbf{P}\left[B = \frac{1}{2} \mid X + Y = 4\right]$ $\mathcal{L}[B \mid X + Y]$	State monad run: $b = [0,1]$ or $b = a \to c \to [0,1]$	$\mathbf{P}[\![\cdot]\!]$, $\mathbf{D}[\![\cdot]\!]$, **`Pr`**, **`Dist`**

Developing the whole semantics right now would take too much time, so I'm going to give some examples of syntax and talk about the structure of the calculations.

First we have random variable expressions. In the first example, X and Y are random variables, so they're functions of Omega. I've already hinted that you could interpret this by regarding random variables as reader monad computations.

But there's no reason to impose a total order, so we use the corresponding applicative functor, or idiom.

Next, we have statements about random variables. A collection of statements is a probabilistic model. We interpret each statement as transforming the global probability space. The first example, X is distributed Geometric B, extends the probability space. The second example is a `condition,' which asserts that applying the random variable X+Y to any world must yield 4. It *restricts* the global probability space.

A nice way to structure these calculations is with the state monad, with probability-space-valued state.

Last, we have queries. The first example is a `conditional probability query'. It conditions the probability space first, and then asks for the probability that B outputs 1/2. The second example is like the first, but is parameterized on the outputs of B and X+Y. It should return a function, or a distribution, so it's a `distribution query'.

Queries run the probability space monad computation in their own particular way.

Interpreting Notation

Syntactic Category	**Examples**	**Computational Structure**	**Semantic Functions**
Expressions	$X + Y$ $\mathrm{Geometric}(B)$	Environment idiom: $R\ a = \Omega \to a$	$\mathcal{R}[\![\cdot]\!]$, **`RV`**
Statements	$X \sim \mathrm{Geometric}(B)$ $X + Y = 4$	State monad: $M\ b = PS \to (PS, b)$ (usually $b = R\ a$)	$\mathcal{M}[\![\cdot]\!]$, **`model`**
Queries	$\mathbf{P}\left[B = \frac{1}{2} \mid X + Y = 4\right]$ $\mathcal{L}[B \mid X + Y]$	State monad run: $b = [0, 1]$ or $b = a \to c \to [0, 1]$	$\mathbf{P}[\![\cdot]\!]$, $\mathbf{D}[\![\cdot]\!]$, **`Pr`**, **`Dist`**

- Difficult to encode types in most type systems

Now, you might think that with all these types and idioms and monads and such, Racket might not be the best implementation language. But Racket's macro system allows us to implement the semantic functions directly. Besides, these types are difficult to encode in most type systems; it seems to require either unityped random variables or dependent types. It's possible in Typed Racket using occurrence typing, but it's a little too much trouble.

Fun Facts: Semantics

- $\mathcal{R}[\![\cdot]\!] : \lambda_{\mathrm{ZFC}} \rightarrow \lambda_{\mathrm{ZFC}}$ interprets anything constructive
  - Uncountable Ω: need to prove measurability conditions

The random variable expression semantic function can turn any lambda-ZFC expression into random variable. When Omega is uncountable, we're going to have to start worrying about something called measurability, but we'll be able to worry about it compositionally.

Fun Facts: Semantics

- $\mathcal{R}[\![\cdot]\!] : \lambda_{\mathrm{ZFC}} \to \lambda_{\mathrm{ZFC}}$ interprets anything constructive
  - Uncountable Ω: need to prove measurability conditions
- $\mathcal{R}[\![\mathrm{Geometric}(B)]\!] : \Omega \to \mathbb{N} \to [0,1]$ is a discrete *transition kernel*
  - Uncountable Ω already works: $\mathcal{R}[\![\mathrm{Normal}(M,S)]\!] : \Omega \to \mathcal{B}(\mathbb{R}) \to [0,1]$

Next is a fortunate accident: the random variable semantic function turns notation that denotes conditional distributions into `transition kernels,' which measure-theoretic probability uses to build probability spaces. So the semantics turns Bayesian notation into exactly what measure-theoretic probability requires, and that fact doesn't change when Omega is uncountable. It also allows Bayesians more freedom: any expression with the right type can be a conditional distribution. I'll show an example later.

Fun Facts: Semantics

- $\mathcal{R}[\![\cdot]\!] : \lambda_{\mathrm{ZFC}} \to \lambda_{\mathrm{ZFC}}$ interprets anything constructive
  - Uncountable Ω: need to prove measurability conditions
- $\mathcal{R}[\![\mathrm{Geometric}(B)]\!] : \Omega \to \mathbb{N} \to [0,1]$ is a discrete *transition kernel*
  - Uncountable Ω already works: $\mathcal{R}[\![\mathrm{Normal}(M,S)]\!] : \Omega \to \mathcal{B}(\mathbb{R}) \to [0,1]$
- Queries approximate with $\mathbf{finitize}\ (\Omega, P)\ k = (\Omega_k, P_k)$
  - Uncountable Ω: $\mathbf{finitize}\ (\Omega, \Sigma, \mathbb{P})\ k = (\Omega_k, P_k)$ with Ω_k finite, stochastic

There's a single point of approximation in the approximating semantics: right before a query, `finitize' restricts Omega to a finite subset of size k. Then, as k approaches infinity, the answer to any query approaches the correct value. Finitize also renormalizes P so that it sums to 1.

Approximations for uncountable Omega are going to be tricky, but there are a lot of available approximations. The most efficient ones are randomized algorithms.

Fun Facts: Implementation

- Almost a transliteration of approximating semantics, except

The implementation is almost a transliteration of the approximating semantics, with the substitutions you would expect; for example,

Fun Facts: Implementation

- Almost a transliteration of approximating semantics, except
  - Lazy lists represent recursively enumerable sets
  - Floats and exact rationals represent probabilities

lazy lists instead of recursively enumerable sets, and floats and rationals instead of reals.

Fun Facts: Implementation

- Almost a transliteration of approximating semantics, except
  - Lazy lists represent recursively enumerable sets
  - Floats and exact rationals represent probabilities
- **RV** `: kstx -> kstx` interprets any Racket expression

Just as the random variable semantic function interprets any lambda-ZFC expression as a random variable, the implementation interprets any Racket expression as a random variable. It does this by fully expanding the expressions first, and then transforming kernel syntax.

Fun Facts: Implementation

- Almost a transliteration of approximating semantics, except
  - Lazy lists represent recursively enumerable sets
  - Floats and exact rationals represent probabilities
- `RV : kstx -> kstx` interprets any Racket expression
- `(define-model name [X ~ ...] ...)` is hygienically referred to by `(with-model name (Pr ... X ...))`

And because Racket has a very expressive macro system, we can separate the monadic computations from the queries that run them, and allow the queries access to the bound identifiers. They can even be in separate modules. This is important for Bayesians, who usually pose many queries for each model.

Duelling Idiots (Paul Nahin)

Let's see how the implementation does on a good, countably infinite probability problem. (By `good,' by the way, I mean that the problem includes gambling and death.) It comes from Paul Nahin's book of puzzlers. Two idiots decide to duel, but they have only one gun, a six-shooter. So they put a bullet in it and take turns spinning the chamber and firing at each other. What's the probability that the player that shoots first wins?

Duelling Idiots (Paul Nahin)

```
(define-model idiot-duel
  [winning-shot ~ (Geometric 1/6)])
```


The trick to answering the query is to recognize that how many shots it takes before the gun finally goes off has a geometric distribution.

Duelling Idiots (Paul Nahin)

```
(define-model idiot-duel
  [winning-shot ~ (Geometric 1/6)])

(with-model idiot-duel
  (Pr (odd? winning-shot)))
; --> 2/3 as k --> ∞
```


The probability that player one wins is the probability that the winning shot is odd-numbered, and this approaches 2/3 as k approaches infinity. So player one has a much better chance of winning this duel. But suppose the idiots know this, so they come up with a plan to even the odds. Player one takes one shot, then player two takes two shots, player one takes three shots, and so on. What's the probability that player one wins?

Duelling Idiots (Paul Nahin)

```
(define-model idiot-duel
  [winning-shot ~ (Geometric 1/6)])

(with-model idiot-duel
  (Pr (odd? winning-shot)))
; --> 2/3 as k --> ∞

(with-model idiot-duel
  (Pr (p1-fires? winning-shot)))
```


Our query now looks like this, where p1-fires? is defined by

Duelling Idiots (Paul Nahin)

```
(define-model idiot-duel
  [winning-shot ~ (Geometric 1/6)])

(with-model idiot-duel
  (Pr (odd? winning-shot)))
; --> 2/3 as k --> ∞

(with-model idiot-duel
  (Pr (p1-fires? winning-shot)))

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


Designing p1-fires? was the trickiest part of the solution. Don't stare at it too long, though; the point is that it exists and isn't too hard to write.

Duelling Idiots (Paul Nahin)

```
(define-model idiot-duel
  [winning-shot ~ (Geometric 1/6)])

(with-model idiot-duel
  (Pr (odd? winning-shot)))
; --> 2/3 as k --> ∞

(with-model idiot-duel
  (Pr (p1-fires? winning-shot)))

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

Nahin (MATLAB):	0.5239191275550995247919843
Us (Racket, k=321):	0.52391912755509952479198439

Nahin spends a page of his book describing his MATLAB solution, which uses problem transformation and symbolic algebra hackery. He computes the answer to 25 decimal places. Our solution consists of just the declarative encoding of the problem on this slide, which took five minutes to write and test. We get the same first 25 digits, but the 26th is 9... so it looks like Nahin should have rounded up.

Duelling Idiot and Half-Wit

But the probelm isn't Bayesian! So suppose that player one is actually a half-wit, and proposes flipping a coin to see whether they will spin the chamber. If they don't spin it, the gun will go off within six shots, and for four of those shots, it will be in player one's hand. But player two is an idiot and agrees to it.

Duelling Idiot and Half-Wit

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


The model looks like this now. The boolean-valued random variable spin? represents the coin flip. Winning-shot's conditional distribution is specified using cond. A Bayesian would normally write his own first-order function instead. This is much nicer, and it has a precise meaning because we have a compositional semantics.

Duelling Idiot and Half-Wit

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

 (with-model half-wit-duel
   (Pr spin? (not (p1-fires? winning-shot))))
```


The probability that player one wins isn't really a Bayesian question. But this is: what's the probability that they spun the chamber given that player two won?

Duelling Idiot and Half-Wit

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

 (with-model half-wit-duel
   (Pr spin? (not (p1-fires? winning-shot))))
```

Answer: about 0.588 (compare `(Pr spin?)` = 0.5)

And the answer is... a little bit more than 1/2. Knowing just the outcome of the duel tells us a little bit about its causes... and that's Bayesian.

Observational Equivalence

- Model equivalence: $m \equiv_{\mathbf{D}} m'$ means no query q can distinguish between m and m'

Because we have a semantics, we can define a notion of observational equivalence, which lets us determine when we can perform optimizations...

Observational Equivalence

- Model equivalence: $m \equiv_{\mathbf{D}} m'$ means no query q can distinguish between m and m'
- Justifies measure-theoretic optimizations
  - Variable collapse (constant folding for rvs)

$$X \sim \mathrm{Normal}(0,1);\ Y \sim \mathrm{Normal}(X,1) \ \longrightarrow\ Y \sim \mathrm{Normal}(0,2)$$

  - Propagating conditions (like constraint propagation)

$$X \sim P_X;\ \ldots;\ X = 3 \ \longrightarrow\ X \sim P_X;\ X = 3;\ \ldots$$

... like variable collapse, which is like constant folding, and condition propagation, which is like constraint propagation. Both of them can yield order-of-magnitude speedups.

Observational Equivalence

- Model equivalence: $m \equiv_{\mathbf{D}} m'$ means no query q can distinguish between m and m'
- Justifies measure-theoretic optimizations
  - Variable collapse (constant folding for rvs)

$$X \sim \mathrm{Normal}(0,1);\ Y \sim \mathrm{Normal}(X,1) \ \longrightarrow\ Y \sim \mathrm{Normal}(0,2)$$

  - Propagating conditions (like constraint propagation)

$$X \sim P_X;\ \ldots;\ X = 3 \ \longrightarrow\ X \sim P_X;\ X = 3;\ \ldots$$

- Justifiable **only** in the **exact semantics**

And both of them can only be proven correct in the exact semantics.

Observational Equivalence

- Model equivalence: $m \equiv_{\mathbf{D}} m'$ means no query q can distinguish between m and m'
- Justifies measure-theoretic optimizations
  - Variable collapse (constant folding for rvs)

$$X \sim \mathrm{Normal}(0,1);\ Y \sim \mathrm{Normal}(X,1) \ \longrightarrow\ Y \sim \mathrm{Normal}(0,2)$$

  - Propagating conditions (like constraint propagation)

$$X \sim P_X;\ \ldots;\ X = 3 \ \longrightarrow\ X \sim P_X;\ X = 3;\ \ldots$$

- Justifiable **only** in the **exact semantics**
  - Suppose for $k = 29$, $q\ m = 0.7$ but $q\ m' = 0.2$

Suppose that, at the 29th approximation, a certain query returned 0.7. Then, after an `optimization,' the same query returned 0.2. It's obviously wrong.

Observational Equivalence

- Model equivalence: $m \equiv_{\mathbf{D}} m'$ means no query q can distinguish between m and m'
- Justifies measure-theoretic optimizations
  - Variable collapse (constant folding for rvs)

$$X \sim \text{Normal}(0,1);\ Y \sim \text{Normal}(X,1) \ \longrightarrow\ Y \sim \text{Normal}(0,2)$$

  - Propagating conditions (like constraint propagation)

$$X \sim P_X;\ \ldots;\ X = 3 \ \longrightarrow\ X \sim P_X;\ X = 3;\ \ldots$$

- Justifiable **only** in the **exact semantics**
  - Suppose for $k = 29$, $q\ m = 0.7$ but $q\ m' = 0.2$
  - But what if, for $k = 400$, $q\ m = 0.19$?

But then, what if we find that at the 400th approximation, the original query returns 0.19? Maybe the optimization preserves meaning and speeds convergence.

Observational Equivalence

- Model equivalence: $m \equiv_{\mathbf{D}} m'$ means no query q can distinguish between m and m'
- Justifies measure-theoretic optimizations
  - Variable collapse (constant folding for rvs)

$$X \sim \text{Normal}(0,1);\ Y \sim \text{Normal}(X,1) \ \longrightarrow\ Y \sim \text{Normal}(0,2)$$

  - Propagating conditions (like constraint propagation)

$$X \sim P_X;\ \ldots;\ X = 3 \ \longrightarrow\ X \sim P_X;\ X = 3;\ \ldots$$

- Justifiable **only** in the **exact semantics**
  - Suppose for $k = 29$, $q\ m = 0.7$ but $q\ m' = 0.2$
  - But what if, for $k = 400$, $q\ m = 0.19$?

In fact, that's what most measure-theoretic optimizations do. Attempting to reason about them in the approximating semantics, or heaven forbid the implementation, would be a total mess, and we'd end up reconstructing the exact semantics anyway.

Having an exact, compositional semantics sets our work apart from other work on Bayesian modeling languages, and in short, it's an awesome thing to have.