Report on the Probabilistic Language Scheme

Alexey Radul

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

ITA Hacker Talk, Oct 17th, 2007

Outline

Outline

Probability Theory
Reasoning Despite Uncertainty

- Mathematical theory of plausible reasoning
- Rules for concluding strength of belief in consequences from strengths of belief in causes

$$p(A \text{ and } B) = p(A) * p(B|A) = p(B) * p(A|B)$$

- Consequently Bayes Rule

$$p(B|A) = p(B) * p(A|B)/p(A)$$

Probabilistic Inference

Inference is the problem of actually carrying these calculations out

- For some concrete collection of known distributions, and some concrete collection of known evidence, we want to know the distributions that information forces for various related unknowns

Probabilistic Inference is Useful

- for spam filtering, Sahami et al 1998
- for robots driving through deserts, Thurn et al 2006
- for studying gene expression, Segal et al 2001
- and many, many more

. . . but Hard to Use in Practice

- Inference algorithms are complicated, thus hard to write from scratch
- Existing inference systems are a pain to use
  - hew close to their assumptions
  - not modular
  - hard to interoperate with
- Can we design a good domain-specific language for inference?

An Inference Library for Scheme

- an experiment in language design
- explores the tension between programming convenience and allowing efficient inference

If you were going to build your own inference library
how would you do it?

The main questions are:

- How to represent distributions?
- How to create, combine, and manipulate distributions?
- How to get answers out of distributions?

Outline

The Natural Literal Syntax for Distributions
is an association list

```
(make-probability-distribution '(obj1 prob1)
'(obj2 prob2) ...)
```

The Natural Literal Syntax for Distributions
is an association list

For example, the distribution for the results of rolling a fair die:

```
(define die-roll-distribution
  (make-discrete-distribution
   '(1 1/6) '(2 1/6) '(3 1/6)
   '(4 1/6) '(5 1/6) '(6 1/6)))
```

The Natural Combinators for Distributions

follow the laws of probability theory

Forward combination:

$$p(x, y) = p(x) * p(y|x)$$

Life is easier if distributions are always over just one value, so instead use

$$p(f(x, y)) = \sum_{x', y' \text{ with } f(x', y') = f(x, y)} p(x') * p(y'|x')$$

```
(dependent-product distribution conditional
combiner)
```

The Natural Combinators for Distributions

follow the laws of probability theory

Forward combination:

$$p(x, y) = p(x) * p(y|x)$$

Life is easier if distributions are always over just one value, so instead use

$$p(f(x, y)) = \sum_{x', y' \text{ with } f(x', y') = f(x, y)} p(x') * p(y'|x')$$

```
(dependent-product distribution conditional
combiner)
```

The Natural Combinators for Distributions

follow the laws of probability theory

Forward combination:

$$p(x,y) = p(x) * p(y|x)$$

Life is easier if distributions are always over just one value, so instead use

$$p(f(x,y)) = \sum_{x',y' \text{ with } f(x',y')=f(x,y)} p(x') * p(y'|x')$$

```
(dependent-product distribution conditional
combiner)
```

The Natural Combinators for Distributions

follow the laws of probability theory

For example, the distribution for rolling two dice can be constructed by:

```
(define two-die-roll-distribution
  (dependent-product
   die-roll-distribution
   (lambda (result1)
    ; The first die does not affect the second
    die-roll-distribution)
   +)) ; We want the sum of the faces
```

The Natural Combinators for Distributions
follow the laws of probability theory

Backward combination:

$$p(x|A(x)) = \begin{cases} p(x)/p(A) & \text{if } A(x) \text{ is true} \\ 0 & \text{if } A(x) \text{ is false} \end{cases}$$

```
(conditional-distribution distribution
predicate)
```

The Natural Combinators for Distributions

follow the laws of probability theory

For example, if we know someone rolled two dice and got more than 9:

```
(conditional-distribution
 two-die-roll-distribution
 (lambda (sum) (> sum 9)))
```

A Naive Representation
Association lists

Represent distributions as association lists (or hash tables) mapping objects to probabilities

- `conditional-distribution` is a filter followed by a renormalization
- `dependent-product` is straightforward too
- querying can be “what is the probability of this object?”
- iteration can be “run me through all object-probability pairs”

The Problem
Long Tails

Distributions with long tails waste computation on irrelevancies

- Possible parse trees of a sentence
    - There are a vast number of them, but most are extremely unlikely
- How many times will one flip heads on a fair coin before the first tail?
    - Infinite, but again, the tail is probably irrelevant

The Solution, version 1

Streams, in the SICP sense of the word

Represent a distribution as a stream of object-probability pairs

- querying becomes "tell me the upper and lower bounds on the probability of this object"
- iteration becomes "give me the underlying stream"
- also need "please compute some more, to bring the bounds closer together"
- yields a restartable, bounded-error anytime approximation strategy
- but . . .

The Problem with the Solution

Normalization

- If a distribution is a list, you know the sum of the probabilities, and can normalize them
- If it's a stream, you won't know the sum until you get to the end
  - Which may be never
- If you condition, probability will disappear
  - Making earlier objects more likely

The Solution, version 2
Impossibilities

The solution is to allow distinguished impossibilities in the stream

- An "impossibility" represents probability that disappears to an unsatisfied predicate.
- Keep a cache that remembers how much probability is gone and normalizes implicitly when asked about the bounds on the probabilities of various objects.

Outline

Defining Complex Distributions

- `make-discrete-distribution` lets you make simple distributions you already know
- then you can morph them with `dependent-product` and `conditional-distribution`
- but that gets really messy really fast

Defining Complex Distributions

For example, the two dice from before took this code:

```
(define die-roll-distribution
  (make-discrete-distribution
   '(1 1/6) '(2 1/6) '(3 1/6)
   '(4 1/6) '(5 1/6) '(6 1/6)))

(let ((two-die-roll-distribution
       (dependent-product
        die-roll-distribution
        (lambda (result1) die-roll-distribution)
        +)))
  (conditional-distribution
   two-die-roll-distribution
   (lambda (sum) (> sum 9))))
```

Defining Complex Distributions

It would be much nicer to write

```
(define (roll-die)
  (discrete-select
   (1 1/6) (2 1/6) (3 1/6)
   (4 1/6) (5 1/6) (6 1/6)))

(let ((num (+ (roll-die) (roll-die))))
  (observe! (> num 9))
  num)
```

Random Processes are Functions with Choices

Many distributions are naturally described as the results of some random process, perhaps with some additional known information

- Rolling dice is a process
- Objects in the desert cause laser ranging data according to a process
- Gene expression is a process, with some unseen choices, that eventually leads to the observed microarray data
- So you'd like to be able to write these processes down naturally:
    - As functions
    - that make some "unseen" decisions

The Primitives of Choices

- `discrete-select` introduces stochastic choices
- `observe!` constrains previous choices with observations
- `stochastic-thunk->distribution` takes a thunk implementing a random process and returns the distribution on that thunk's return values

Rolling Dice, Again

```
(define (roll-die)
  (discrete-select
   (1 1/6) (2 1/6) (3 1/6)
   (4 1/6) (5 1/6) (6 1/6)))

(stochastic-thunk->distribution
 (lambda ()
   (let ((num (+ (roll-die) (roll-die))))
     (observe! (> num 9))
     num)))
```

Flipping Coins

the easy way

```
(define (num-flips-until-tail)
  (discrete-select
   (0 1/2)
   ((+ 1 (num-flips-until-tail)) 1/2)))

(stochastic-thunk->distribution
 num-flips-until-tail)
```

Flipping Coins
the hard way

```
(define (coin-flipping-distribution)
  (dependent-product
   (make-discrete-distribution
    '(tails 1/2) '(heads 1/2))
   (lambda (symbol)
     (if (eq? symbol 'tails)
         (make-discrete-distribution (list 0 1))
         (coin-flipping-distribution)))
   (lambda (first-flip num-other-flips)
     (if (eq? first-flip 'tails)
         0
         (+ 1 num-other-flips)))))
```

Summary

- Lazy Streams make a good underlying representation for discrete probability distributions
- Stochastic Functions make a good way to define discrete probability distributions