Report on the Probabilistic Language Scheme

Alexey Radul

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

DLS 2007, Oct 22nd

Outline

1. Motivation
   - Background
   - The Problem
   - The Approach
2. Representation
   - Lists?
   - Lazy Streams
3. Interface
   - Explicit Distributions?
   - Implicit Distributions?
   - The Answer

Probability theory exists:

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

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

Probabilistic inference is useful

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

. . . but hard to use

- algorithms are complicated
- existing systems are a pain to use
  - hew close to their assumptions
  - not modular
  - hard to interoperate with

Can we do better?

We can try

- library for Scheme
- experiment in language design

We can try

- library for Scheme
- experiment in language design

First big question:

Representation?

Lists?

Lists lose on long tails

Long tails

- 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
- and many, many more

So?

Lazy Streams

Lazy Streams

- Delay computing the long tail
- You likely won't need it anyway

Approximation,

but the best kind:

- anytime
- restartable
- bounded-error

Approximation,

but the best kind:

- anytime
- restartable
- bounded-error

There's also some fine print

The streams need to allow duplicates.
The streams need to allow explicit statements of impossibility.
The objects exiting the streams need to be cached.
The caches need to be kept up to date
Even in the face of aliasing and direct access to the streams.
If you really want to know, ask during the question period.

There's also some fine print

The streams need to allow duplicates.
The streams need to allow explicit statements of impossibility.
The objects exiting the streams need to be cached.
The caches need to be kept up to date
Even in the face of aliasing and direct access to the streams.
If you really want to know, ask during the question period.

But it all works out

Second big question:

API?

Explicit Distribution Objects?

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

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

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

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

Looks ok, but...

```
(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))))
```

Instead:

```
(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)
```

Implicit Distributions?

```
(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)
```

Querying? Modularity?

Answer:

Both!

```
(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)))
```

Contributions

- Representation: Lazy Streams
- API: Stochastic Functions AND Explicit Objects

Another example:

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 '(0 1))
         (coin-flipping-distribution)))
   (lambda (first-flip num-further-flips)
     (if (eq? first-flip 'tails)
         0
         (+ 1 num-further-flips)))))
```