

Probabilistic Functional Programming

Donnacha Oisín Kidney

July 17, 2018

Modeling Probability

An Example

Unclear Semantics

Underpowered

Monadic Modeling

The Erwig And

Kollmansberger Approach

Other Interpreters

Theoretical Foundations

Stochastic Lambda Calculus

Giry Monad

Other Applications

Differential Privacy

Conclusion

Modeling Probability

How do we model stochastic and probabilistic processes in programming languages?

The Boy-Girl Paradox

1. Mr. Jones has two children. The older child is a girl. What is the probability that both children are girls?
2. Mr. Smith has two children. At least one of them is a boy. What is the probability that both children are boys?

Probabilistic Functional Programming

└ Modeling Probability

└ An Example

└ The Boy-Girl Paradox

1. Mr. Jones has two children. The older child is a girl. What is the probability that both children are girls?
2. Mr. Smith has two children. At least one of them is a boy. What is the probability that both children are boys?

Gardner originally wrote that the second question (perhaps surprisingly) has the answer $1/3$. However, he later acknowledged the question was ambiguous, and agreed that certain interpretations could correctly conclude its answer was $1/2$

The Boy-Girl Paradox

1. Mr. Jones has two children. The older child is a girl. What is the probability that both children are girls?
2. Mr. Smith has two children. At least one of them is a boy. What is the probability that both children are boys?

Is the answer to 2 $\frac{1}{3}$ or $\frac{1}{2}$?

Probabilistic Functional Programming

└ Modeling Probability

└ An Example

└ The Boy-Girl Paradox

1. Mr. Jones has two children. The older child is a girl. What is the probability that both children are girls?
2. Mr. Smith has two children. At least one of them is a boy. What is the probability that both children are boys?

Is the answer to 2 $\frac{1}{3}$ or $\frac{1}{2}$?

Gardner originally wrote that the second question (perhaps surprisingly) has the answer $1/3$. However, he later acknowledged the question was ambiguous, and agreed that certain interpretations could correctly conclude its answer was $1/2$

The Boy-Girl Paradox

1. Mr. Jones has two children. The older child is a girl. What is the probability that both children are girls?
2. Mr. Smith has two children. At least one of them is a boy. What is the probability that both children are boys?

Is the answer to 2 $\frac{1}{3}$ or $\frac{1}{2}$?

Part of the difficulty in the question is that it's ambiguous: can we use programming languages to lend some precision?

Probabilistic Functional Programming

└ Modeling Probability

└ An Example

└ The Boy-Girl Paradox

1. Mr. Jones has two children. The older child is a girl. What is the probability that both children are girls?
2. Mr. Smith has two children. At least one of them is a boy. What is the probability that both children are boys?

Is the answer to 2 $\frac{1}{3}$ or $\frac{1}{2}$?

Part of the difficulty in the question is that it's ambiguous: can we use programming languages to lend some precision?

Gardner originally wrote that the second question (perhaps surprisingly) has the answer $1/3$. However, he later acknowledged the question was ambiguous, and agreed that certain interpretations could correctly conclude its answer was $1/2$

An Ad-Hoc Solution i

Using normal features built in to the language.

```
from random import randrange, choice

class Child:
    def __init__(self):
        self.gender = choice(['boy', 'girl'])
        self.age = randrange(18)
```

```
from operator import attrgetter

def mr_jones():
    child_1 = Child()
    child_2 = Child()
    eldest = max(child_1, child_2,
                  key=attrgetter('age'))
    assert eldest.gender == 'girl'
    return [child_1, child_2]
```

```
def mr_smith():  
    child_1 = Child()  
    child_2 = Child()  
    assert child_1.gender == 'boy' or \  
           child_2.gender == 'boy'  
    return [child_1, child_2]
```

Unclear semantics

What contracts are guaranteed by probabilistic functions? What does it mean *exactly* for a function to be probabilistic? Why isn't the following¹ “random”?

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
             // guaranteed to be random.  
}
```

¹Randall Munroe. *Xkcd: Random Number*. en. Title text: RFC 1149.5 specifies 4 as the standard IEEE-vetted random number. Feb. 2007. URL: <https://xkcd.com/221/> (visited on 07/06/2018).

What about this?

```
children_1 = [Child(), Child()]
```

```
children_2 = [Child()] * 2
```

How can we describe the difference between `children_1` and `children_2`?

Probabilistic Functional Programming

- └ Modeling Probability
- └ Unclear Semantics

What about this?

```
children_1 = [Child(), Child()]  
children_2 = [Child()] * 2
```

How can we describe the difference between `children_1` and `children_2`?

The first runs two random processes; the second only one. Both have the same types, both look like they do the same thing. We need a good way to describe the difference between them.

Underpowered

There are many more things we may want to do with probability distributions.

What about expectations?

```
def expect(predicate, process, iterations=100):  
    success, tot = 0, 0  
    for _ in range(iterations):  
        try:  
            success += predicate(process())  
            tot += 1  
        except AssertionError:  
            pass  
    return success / tot
```

2018-07-17

Probabilistic Functional Programming

- └ Modeling Probability

- └ Underpowered

- └ Underpowered

Underpowered

There are many more things we may want to do with probability distributions.

What about expectations?

```
def expect(predicate, process, iterations=100):
    success, tot = 0, 0
    for _ in range(iterations):
        try:
            success += predicate(process())
            tot += 1
        except AssertionError:
            pass
    return success / tot
```

This solution is both inefficient and inexact. Also, we may want to express other attributes of probability distributions: independence, for example.

The Ad-Hoc Solution

```
p_1 = expect(  
    lambda children: all(child.gender == 'girl'  
                           for child in children),  
    mr_jones)  
p_2 = expect(  
    lambda children: all(child.gender == 'boy'  
                           for child in children),  
    mr_smith)
```

$$p_1 \cong \frac{1}{2}$$

$$p_2 \cong \frac{1}{3}$$

Monadic Modeling

What we're approaching is a DSL, albeit an unspecified one.

A DSL

What we're approaching is a DSL, albeit an unspecified one.

Three questions for this DSL:

What we're approaching is a DSL, albeit an unspecified one.

Three questions for this DSL:

- Why should we implement it? What is it useful for?

What we're approaching is a DSL, albeit an unspecified one.

Three questions for this DSL:

- Why should we implement it? What is it useful for?
- How should we implement it? How can it be made efficient?

What we're approaching is a DSL, albeit an unspecified one.

Three questions for this DSL:

- Why should we implement it? What is it useful for?
- How should we implement it? How can it be made efficient?
- Can we glean any insights on the nature of probabilistic computations from the language? Are there any interesting symmetries?

The Erwig And Kollmansberger Approach

First approach²: `newtype Dist a = Dist runDist :: [(a, Rational)]`
A distribution is a list of possible events, each tagged with a probability.

²Martin Erwig and Steve Kollmansberger. "Functional Pearls: Probabilistic Functional Programming in Haskell". In: *Journal of Functional Programming* 16.1 (2006), pp. 21–34. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796805005721. URL: <http://web.engr.oregonstate.edu/~erwig/papers/abstracts.html%5C#JFP06a> (visited on 09/29/2016).

Probabilistic Functional Programming

└ Monadic Modeling

└ The Erwig And Kollmansberger Approach

└ The Erwig And Kollmansberger Approach

First approach²: `newtype Dist a = Dist runDist :: [(a, Rational)]`
A distribution is a list of possible events, each tagged with a probability.

²Martin Erwig and Steve Kollmansberger, "Functional Pearls: Probabilistic Functional Programming in Haskell", in: *Journal of Functional Programming* 16.1 (2006), pp. 21–54. [doi: 10.1017/S0095679606000072](https://doi.org/10.1017/S0095679606000072); [url: http://web.eecs.oregonstate.edu/~erwig/papers/abstracts.html#SKW06a](http://web.eecs.oregonstate.edu/~erwig/papers/abstracts.html#SKW06a) (visited on 06/28/2018).

This representation only works for discrete distributions

We could (for example) encode a die as: $\text{die} :: \text{Dist Integer die} = \text{Dist} [(1, \frac{1}{6}), (2, \frac{1}{6}), (3, \frac{1}{6}), (4, \frac{1}{6}), (5, \frac{1}{6}), (6, \frac{1}{6})]$

This lets us encode (in the types) the difference between:
 $children_1 :: [DistChild] \quad children_2 :: Dist[Child]$

As we will use this as a DSL, we need to define the language features we used above:

```
def mr_smith():  
    child_1 = Child()  
    child_2 = Child()  
    assert child_1.gender == 'boy' or \  
           child_2.gender == 'boy'  
    return [child_1, child_2]
```

As we will use this as a DSL, we need to define the language features we used above:

```
def mr_smith():  
    child_1 = Child()  
    child_2 = Child()  
    assert child_1.gender == 'boy' or \  
           child_2.gender == 'boy'  
    return [child_1, child_2]
```

1. = (assignment)

As we will use this as a DSL, we need to define the language features we used above:

```
def mr_smith():  
    child_1 = Child()  
    child_2 = Child()  
    assert child_1.gender == 'boy' or \  
           child_2.gender == 'boy'  
    return [child_1, child_2]
```

1. = (assignment)
2. assert

As we will use this as a DSL, we need to define the language features we used above:

```
def mr_smith():  
    child_1 = Child()  
    child_2 = Child()  
    assert child_1.gender == 'boy' or \  
           child_2.gender == 'boy'  
    return [child_1, child_2]
```

1. = (assignment)
2. assert
3. return

Assignment 1

Assignment expressions can be translated into lambda expressions:
 $\text{let } x = e_1 \text{ in } e_2 == (\lambda x. e_2) e_1$ In the context of a probabilistic language, e_1 and e_2 are distributions. So what we need to define is application: this is encapsulated by the “monadic bind”: $(\lambda x. e) :: \text{Dist } a \rightarrow (a \rightarrow \text{Dist } b) \rightarrow \text{Dist } b$ For a distribution, what’s happening inside the λ is e_1 given x . Therefore, the resulting probability is the product of the outer and inner probabilities. $\lambda x. e = f = \text{Dist } [(y, x \cdot p * y_p) \mid (x, p) \leftarrow \text{runDist } xs, (y, p_p) \leftarrow \text{runDist } (f \ x)]$

Assertion is a kind of conditioning: given a statement about an event, it either occurs or it doesn't. $\text{guard} :: \text{Bool} \rightarrow \text{Dist } ()$
 $\text{guard True} = \text{Dist } [(), 1]$ $\text{guard False} = \text{Dist } []$

Return is the “unit” value for a distribution; the certain event, the unconditional distribution. $\text{return} :: a \rightarrow \text{Dist } a$
 $\text{return } x = \text{Dist } [(x, 1)]$

Putting it all Together

```
mrSmith :: Dist [Child]
mrSmith = do child1 <- child
              child2 <- child
              guard (gender child1 == Boy && gender child2 == Boy)
              return [child1, child2]
```

```
expect :: (a -> Rational) -> Dist a -> Rational
expect p xs = frac
  (sum [ p x * xp | (x,xp) <- runDist xs ])
  (sum [ xp | (x,xp) <- runDist xs ])
```

```
probOf :: (a -> Bool) -> Dist a -> Rational
probOf p = expect (\x -> if p x then 1 else 0)
```

```
probOf (all ((==) Girl . gender)) mrJones == frac 1 2 probOf (all  
((==) Boy . gender)) mrSmith == frac 1 3
```

Alternative Interpreters

Once the semantics are described, different interpreters are easy to swap in.

```
data Decision = Decision stick :: Bool , switch :: Bool
montyHall :: Dist Decision
montyHall = do car ← uniform [1..3]
choice1 <- uniform [1..3]
let left = [door | door < -[1..3], door /= choice1]
let open = head [door | door < -left, door /= car]
let choice2 = head [door | door < -left, door /= open]
return (Decision stick == car == choice1, switch = car == choice2)
```


While we can interpret it in the normal way to solve the problem:
 $\text{probOf stick montyHall} == \text{frac } 1 \ 3$ $\text{probOf switch montyHall} ==$
 $\text{frac } 2 \ 3$

Monty Hall iii

We could alternatively draw a diagram of the process.

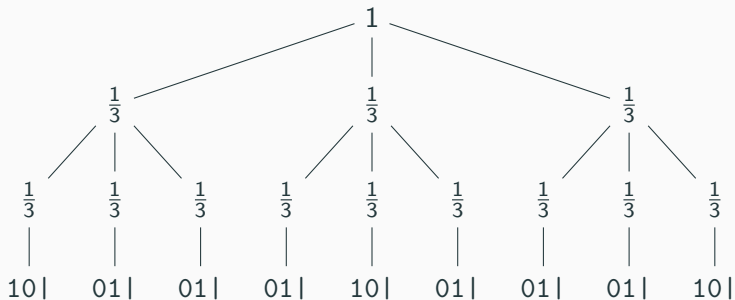


Figure 1: AST from Monty Hall problem. $1|$ is a win, $0|$ is a loss. The first column is what happens on a stick, the second is what happens on a loss.

Theoretical Foundations

Stochastic Lambda Calculus

It is possible³ to give measure-theoretic meanings to the operations described above.

$$\mathcal{M} \text{ return } x(A) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

$$\mathcal{M} dk(A) = \int_X \mathcal{M} k(x)(A) d\mathcal{M} d(x) \quad (2)$$

³Norman Ramsey and Avi Pfeffer. “Stochastic Lambda Calculus and Monads of Probability Distributions”. In: *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Vol. 37. ACM, 2002, pp. 154–165.

URL: <http://www.cs.tufts.edu/~nr/cs257/archive/norman-ramsey/pmonad.pdf> (visited on 09/29/2016).

Probabilistic Functional Programming

- └ Theoretical Foundations
 - └ Stochastic Lambda Calculus
 - └ Stochastic Lambda Calculus

Stochastic Lambda Calculus

It is possible³ to give measure-theoretic meanings to the operations described above.

$$M \text{ return } x(A) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

$$M \text{ do}(A) = \int_X M \lambda(x)(A) dM \, d(x) \quad (2)$$

³Norman Ramsey and Avi Pfeffer, "Stochastic Lambda Calculus and Monads of Probability Distributions", in: 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Vol. 37, ACM, 2002, pp. 154–165.
 URL: <http://www.cs.tufts.edu/~nfr/cx2007/archives/normramsey/papersad.pdf> (visited on 09/28/2016).

return is the Dirac measure

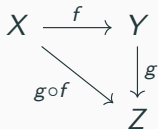
Giry⁴ gave a categorical interpretation of probability theory.

⁴Michèle Giry. “A Categorical Approach to Probability Theory”. In: *Categorical Aspects of Topology and Analysis*. Ed. by A. Dold, B. Eckmann, and B. Banaschewski. Vol. 915. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 68–85. ISBN: 978-3-540-11211-2 978-3-540-39041-1. DOI: 10.1007/BFb0092872. URL: <http://link.springer.com/10.1007/BFb0092872> (visited on 03/03/2017).

Categories, Quickly

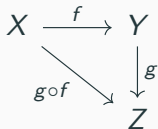
$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ & \searrow g \circ f & \downarrow g \\ & & Z \end{array}$$

Categories, Quickly



Objects $\text{Ob}(\mathbf{C}) = \{X, Y, Z\}$

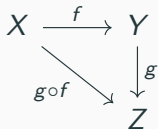
Categories, Quickly



Objects $\text{Ob}(\mathbf{C}) = \{X, Y, Z\}$

Arrows $\text{hom}_{\mathbf{C}}(X, Y) = X \rightarrow Y$

Categories, Quickly

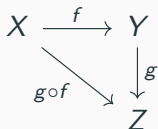


Objects $\text{Ob}(\mathbf{C}) = \{X, Y, Z\}$

Arrows $\text{hom}_{\mathbf{C}}(X, Y) = X \rightarrow Y$

Composition \circ

Categories, Quickly

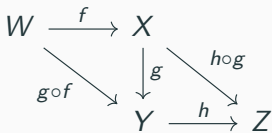


Objects $\text{Ob}(\mathbf{C}) = \{X, Y, Z\}$

Arrows $\text{hom}_{\mathbf{C}}(X, Y) = X \rightarrow Y$

Composition \circ

Arrows form a monoid under composition

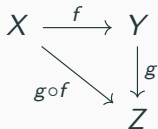


$$(h \circ g) \circ f = h \circ (g \circ f) \quad (3)$$

$$A \curvearrowright id_A$$

$$\forall A. A \in \mathbf{Ob}(\mathbf{C}) \exists id_A : \text{hom}_{\mathbf{C}}(A, A) \quad (4)$$

Categories, Quickly

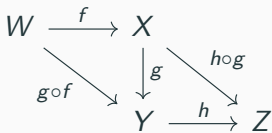


Objects $\text{Ob}(\mathbf{C}) = \{X, Y, Z\}$

Arrows $\text{hom}_{\mathbf{C}}(X, Y) = X \rightarrow Y$

Composition \circ

Arrows form a monoid under composition



$$(h \circ g) \circ f = h \circ (g \circ f) \quad (3)$$

$$A \curvearrowright id_A$$

$$\forall A. A \in \mathbf{Ob}(\mathbf{C}) \exists id_A : \text{hom}_{\mathbf{C}}(A, A) \quad (4)$$

Example

Set is the category of sets, where objects are sets, and arrows are functions.

The category of (small) categories, **Cat**, has morphisms called Functors.

Functors

The category of (small) categories, **Cat**, has morphisms called Functors.

These can be thought of as ways to “embed” one category into another.

Functors

The category of (small) categories, **Cat**, has morphisms called Functors.

These can be thought of as ways to “embed” one category into another.

$$\begin{array}{ccc} \mathbf{F}X & \xrightarrow{\mathbf{F}f} & \mathbf{F}Y \\ \uparrow & & \uparrow \\ X & \xrightarrow{f} & Y \end{array}$$

Functors which embed categories into themselves are called Endofunctors.

Monads

In the category of Endofunctors, **Endo**, a Monad is a triple of:

1. An Endofunctor m ,
2. A natural transformation:

$$\eta : A \rightarrow m(A) \tag{5}$$

This is an operation which embeds an object.

3. Another natural transformation:

$$\mu : m^2(A) \rightarrow m(A) \tag{6}$$

This collapses two layers of the functor.

The Category of Measurable Spaces

Meas is the category of measurable spaces.

The Category of Measurable Spaces

Meas is the category of measurable spaces.

The arrows (**hom**_{Meas}) are measurable maps.

The Category of Measurable Spaces

Meas is the category of measurable spaces.

The arrows (**hom**_{Meas}) are measurable maps.

The objects are measurable spaces.

The Category of Measurable Spaces

Meas is the category of measurable spaces.

The arrows ($\mathbf{hom}_{\mathbf{Meas}}$) are measurable maps.

The objects are measurable spaces.

We can construct a functor (\mathcal{P}), which, for any given measurable space \mathcal{M} , is the space of all possible measures on it.

The Category of Measurable Spaces

Meas is the category of measurable spaces.

The arrows (**hom**_{Meas}) are measurable maps.

The objects are measurable spaces.

We can construct a functor (\mathcal{P}), which, for any given measurable space \mathcal{M} , is the space of all possible measures on it.

$\mathcal{P}(\mathcal{M})$ is itself a measurable space: measuring is integrating over some variable a in \mathcal{M} .

The Category of Measurable Spaces

Meas is the category of measurable spaces.

The arrows (**hom**_{Meas}) are measurable maps.

The objects are measurable spaces.

We can construct a functor (\mathcal{P}), which, for any given measurable space \mathcal{M} , is the space of all possible measures on it.

$\mathcal{P}(\mathcal{M})$ is itself a measurable space: measuring is integrating over some variable a in \mathcal{M} .

In code (we restrict to measurable functions): `newtype Measure a = Measure ((a -> Rational) -> Rational)`

We now get η and μ :

$\text{integrate} :: \text{Measure } a \rightarrow (\text{a} \rightarrow \text{Rational}) \rightarrow \text{Rational}$
 $\text{integrate (Measure m)} f = m f$

$\text{return} :: a \rightarrow \text{Measure } a$
 $\text{return } x = \text{Measure } (\lambda \text{ measure } x)$

$(\lambda \lambda =) :: \text{Measure } a \rightarrow (\text{a} \rightarrow \text{Measure } b) \rightarrow \text{Measure } b$
 $\text{xs} \rightarrow \lambda \lambda = f = \text{Measure } (\lambda \text{ integrate xs } (\lambda \text{ integrate } (f x) (\lambda \text{ measure } y)))$

Other Applications

It has been shown⁵ that the semantics of the probability monad suitable encapsulate *differential privacy*.

⁵Jason Reed and Benjamin C. Pierce. “Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy”. In: *ACM Sigplan Notices*. Vol. 45. ACM, 2010, pp. 157–168. URL: <http://dl.acm.org/citation.cfm?id=1863568> (visited on 03/01/2017).

LINQ⁶ is an API which provides a monadic syntax for performing queries (sql, etc.)

PINQ⁷ extends this to provide *differentially private* queries.

⁶Don Box and Anders Hejlsberg. *LINQ: .NET Language Integrated Query*. en. Feb. 2007. URL:

<https://msdn.microsoft.com/en-us/library/bb308959.aspx> (visited on 07/09/2018).

⁷Frank McSherry. "Privacy Integrated Queries". In: *Communications of the ACM* (Sept. 2010). URL: <https://www.microsoft.com/en-us/research/publication/privacy-integrated-queries-2/>.

Conclusion
