# Probabilistic Functional Programming

Donnacha Oisín Kidney

July 17, 2018

# 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?

# 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 $\frac{1}{2}$?

## 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?

# 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)
```

## An Ad-Hoc Solution ii

```
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]
```

# An Ad-Hoc Solution iii

```python
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 following[1] "random"?

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

[1] Randall 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`?

# 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
```

# 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)
```

$$\texttt{p\_1} \approxeq \frac{1}{2}$$
$$\texttt{p\_2} \approxeq \frac{1}{3}$$

# Monadic Modeling

# A DSL

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:

# A 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?

# A 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?
- How should we implement it? How can it be made efficient?

# A 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?
- 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 approach[2]: newtype Dist a = Dist runDist :: [(a, Rational)]
A distribution is a list of possible events, each tagged with a probability.

[2] Martin 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).

We could (for example) encode a die as: die :: Dist Integer die = 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$_1$ :: $[DistChild] children_2$ :: $Dist[Child]$

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 i

Assignment expressions can be translated into lambda expressions: let x = e$_1 ine_2 == (-- > e_2)e_1$ In the context of a probabilistic language, $e_1$ and $e_1$ are distributions. So what we need to define is application: this is encapsulated by the “monadic bind”: (¿¿=) :: Dist a -¿ (a -¿ Dist b) -¿ Dist b For a distribution, what’s happening inside the $\lambda$ is $e_1$ given $x$. Therefore, the resulting probability is the product of the outer and inner probabilities. xs ¿¿= f = Dist [ (y, xp * yp) — (x, xp) ¡- runDist xs , (y, yp) ¡- runDist (f x) ]

# Assertion

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

# Return

Return is the “unit” value for a distribution; the certain event, the unconditional distribution. return :: a -¿ Dist a return x = 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 — $(_,xp) < -runDistxs])$

probOf :: (a -¿ Bool) -¿ Dist a -¿ Rational probOf p = expect (-¿ 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.

## Monty Hall i

data Decision = Decision stick :: Bool , switch :: Bool montyHall :: Dist Decision montyHall = do car ¡- uniform [1..3] choice$_1 < -\mathit{uniform}[1..3]\mathit{letleft} = [\mathit{door}|\mathit{door} < -[1..3], \mathit{door}/ = \mathit{choice}_1]\mathit{letopen} = \mathit{head}[\mathit{door}|\mathit{door} < -\mathit{left}, \mathit{door}/ = \mathit{car}]\mathit{letchoice}_2 = \mathit{head}[\mathit{door}|\mathit{door} < -\mathit{left}, \mathit{door}/ = \mathit{open}]\mathit{return}(\mathit{Decisionstick} = \mathit{car} == \mathit{choice}_1, \mathit{switch} = \mathit{car} == \mathit{choice}_2)$

## Monty Hall ii

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

We could alternatively draw a diagram of the process.

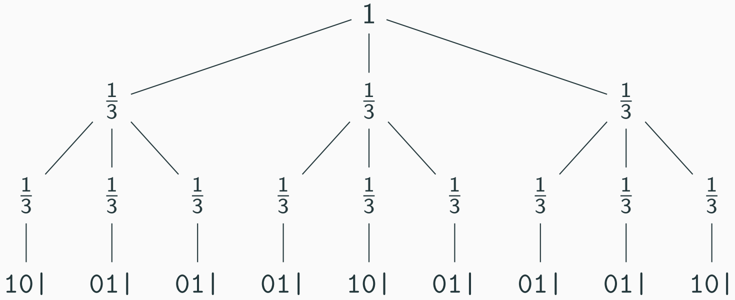


**Figure 1:** AST from Monty Hall problem. 1| is a win, 0| is a loss. The first column is what happens on a stick, the second is what happens on a loss.

# Theoretical Foundations

# Stochastic Lambda Calculus

It is possible[3] to give measure-theoretic meanings to the operations described above.

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

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

[3] Norman 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).

# The Giry Monad

Giry[4] gave a categorical interpretation of probability theory.

[4] Michèle 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

# Categories, Quickly

$$
\begin{tikzcd}
X \arrow[r, "f"] \arrow[dr, "g \circ f"'] & Y \arrow[d, "g"] \\
& Z
\end{tikzcd}
$$

# Categories, Quickly

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[dr, "g \circ f"'] & Y \arrow[d, "g"] \\ & Z \end{tikzcd}$$

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

# Categories, Quickly

$$
\begin{tikzcd}
X \arrow[r, "f"] \arrow[dr, "g \circ f"'] & Y \arrow[d, "g"] \\
& Z
\end{tikzcd}
$$

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

**Arrows** $\mathbf{hom}_{\mathbf{C}}(X, Y) = X \to Y$

# Categories, Quickly

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[dr, "g\circ f"'] & Y \arrow[d, "g"] \\ & Z \end{tikzcd}$$

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

**Arrows** $\mathbf{hom}_{\mathbf{C}}(X, Y) = X \to Y$

**Composition** $\circ$

# Categories, Quickly

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[dr, "g \circ f"'] & Y \arrow[d, "g"] \\ & Z \end{tikzcd}$$

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

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

**Composition** $\circ$

**Arrows form a monoid under composition**

$$\begin{tikzcd} W \arrow[r, "f"] \arrow[dr, "g \circ f"'] & X \arrow[d, "g"] \arrow[dr, "h \circ g"] & \\ & Y \arrow[r, "h"] & Z \end{tikzcd}$$

$$(h \circ g) \circ f = h \circ (g \circ f) \tag{3}$$

$$\begin{tikzcd} A \arrow[loop right, "id_A"] \end{tikzcd}$$

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

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[dr, "g \circ f"'] & Y \arrow[d, "g"] \\ & Z \end{tikzcd}$$

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

**Arrows** $\mathbf{hom_C}(X, Y) = X \to Y$

**Composition** $\circ$

**Arrows form a monoid under composition**

$$\begin{tikzcd} W \arrow[r, "f"] \arrow[dr, "g \circ f"'] & X \arrow[d, "g"] \arrow[dr, "h \circ g"] & \\ & Y \arrow[r, "h"] & Z \end{tikzcd}$$

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

$$\begin{tikzcd} A \arrow[loop right, "id_A"] \end{tikzcd}$$

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

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

# Functors

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:

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

   This is an operation which embeds an object.

3. Another natural transformation:

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

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

# 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.

## The Category of Measurable Spaces

**Meas** is the category of measurable spaces.

The arrows ($\mathbf{hom_{Meas}}$) 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.

# The Category of Measurable Spaces

**Meas** is the category of measurable spaces.

The arrows ($\mathbf{hom_{Meas}}$) 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 ($\mathbf{hom}_{\mathbf{Meas}}$) 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.

$\mathcal{P}(\mathcal{M})$ 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 $\eta$ and $\mu$:

integrate :: Measure a -¿ (a -¿ Rational) -¿ Rational integrate (Measure m) f = m f

return :: a -¿ Measure a return x = Measure (-¿ measure x)

(¿¿=) :: Measure a -¿ (a -¿ Measure b) -¿ Measure b xs ¿¿= f = Measure (-¿ integrate xs (-¿ integrate (f x) (-¿ measure y)))

# Other Applications

# Differential Privacy

It has been shown[5] that the semantics of the probability monad suitable encapsulate *differential privacy*.

[5] Jason 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).

# PINQ

LINQ[6] is an API which provides a monadic syntax for performing queries (sql, etc.)

PINQ[7] extends this to provide *differentially private* queries.

[6] Don 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).

[7] Frank 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