Randomness in Haskell

Tomasz Tylec
November 8, 2018

IF Research Polska / Mazars-Zettafox (France)

tomasz.tylec@ifresearch.pl

Overview

Motivation

Mathematical background

Expressing idea in pseudo-Haskell

The `random-fu` package

Stochastic process

Applications

Motivation

What’s the problem?

- we like pure functions – same input produce same output, no side effects:

```haskell
sq :: Num a => a -> a
sq x = x*x
```

What's the problem?

- we like pure functions – same input produce same output, no side effects:

```haskell
sq :: Num a => a -> a
sq x = x*x
```

- function returning random value cannot be pure

  it is supposed to be random, not deterministic ...

```haskell
data Coin = Head | Tail

-- Flip coin k times
filpCoinK :: Int -> [Coin]  -- can't be random
flipCoinK k = ...
```

What's the problem?

- we like pure functions – same input produce same output, no side effects:

```
sq :: Num a => a -> a
sq x = x*x
```

- function returning random value cannot be pure

  it is supposed to be random, not deterministic ...

```
data Coin = Head | Tail

-- Flip coin k times
filpCoinK :: Int -> [Coin]  -- can't be random
flipCoinK k = ...

flipCoinK' :: Int -> IO [Coin]  -- this could
flipCoinK' k = ...
```

Yes, but ...

- ... but random number generators are deterministic!

```
Coin = Head | Tail

filpCoinK :: Seed -> Int -> (Seed, [Coin])
```

Yes, but ...

- ... but random number generators are deterministic!

```haskell
Coin = Head | Tail

filpCoinK :: Seed -> Int -> (Seed, [Coin])
```

- this requires to pass random generator state everywhere around ...

Yes, but …

- … but random number generators are deterministic!

```
Coin = Head | Tail
```

```
filpCoinK :: Seed -> Int -> (Seed, [Coin])
```

- this requires to pass random generator state everywhere around …
- … we could hide it `State` monad …
- … but still, not so elegant …
- and would not work with true random source.

Yes, but ...

- ... but random number generators are deterministic!

  ```
  Coin = Head | Tail
  ```

  ```
  filpCoinK :: Seed -> Int -> (Seed, [Coin])
  ```

- this requires to pass random generator state everywhere around ...
- ... we could hide it `State` monad ...
- ... but still, not so elegant ...
- and would not work with true random source.

On the other hand, **probability theory** is basically all about randomness.

How did mathematicians get rid of “random outcome of an experiment”?

Mathematical background

Probability space

Let:

- Ω be a set – *sample space* – set of outcomes of some random experiment.

Probability space

Let:

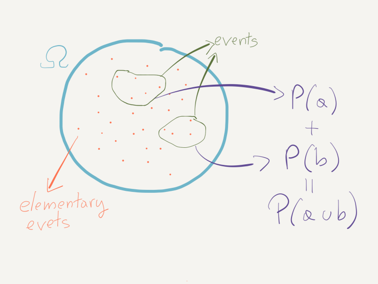
- Ω be a set – *sample space* – set of outcomes of some random experiment.
- $\mathcal{E}$ a “nice” family of subsets of Ω – *set of events* (outcome may not be “sharp”).

Probability space

Let:

- Ω be a set – *sample space* – set of outcomes of some random experiment.
- $\mathcal{E}$ a “nice” family of subsets of Ω – *set of events* (outcome may not be “sharp”).
- $P :: \mathcal{E} \to \mathbb{R}$ a measurable function – *probability measure* – such that:

1. $P(a) \geq 0$ for all $a \in \mathcal{E}$
2. $P(\Omega) = 1$
3. $P(a + b) = P(a) + (b)$ whenever $a \cap b = \emptyset$



Random variable

A (measurable) function $\Omega \to X$ from sample space to some set X. It maps “random experiment outcomes” to some mathematically tractable objects.

e.g. real numbers, but can be other things too.

$\mathcal{R}(\Omega)$ – the set of all random variables on $(\Omega, \mathcal{E}, P)$.

Random variable

A (measurable) function $\Omega \to X$ from sample space to some set X. It maps "random experiment outcomes" to some mathematically tractable objects.

e.g. real numbers, but can be other things too.

$\mathcal{R}(\Omega)$ – the set of all random variables on $(\Omega, \mathcal{E}, P)$.

Expected value

If $r \in \mathcal{R}(\Omega)$ is a random variable, then

$$\mathbb{E}(r) = \int_{\Omega} r(x)\, dp(x)$$

is called expected value of random variable r.

Random variable

A (measurable) function $\Omega \to X$ from sample space to some set X. It maps "random experiment outcomes" to some mathematically tractable objects.

e.g. real numbers, but can be other things too.

$\mathcal{R}(\Omega)$ – the set of all random variables on $(\Omega, \mathcal{E}, P)$.

Expected value

If $r \in \mathcal{R}(\Omega)$ is a random variable, then

$$\mathbb{E}(r) = \int_\Omega r(x)\,dp(x) = \int_\Omega r(x)p(x)\,dx$$

is called expected value of random variable r.

Random variable

A (measurable) function $\Omega \to X$ from sample space to some set X. It maps "random experiment outcomes" to some mathematically tractable objects.

e.g. real numbers, but can be other things too.

$\mathcal{R}(\Omega)$ – the set of all random variables on $(\Omega, \mathcal{E}, P)$.

Expected value

If $r \in \mathcal{R}(\Omega)$ is a random variable, then

$$\mathbb{E}(r) = \int_\Omega r(x)\,dp(x) = \int_\Omega r(x)p(x)\,dx = \sum_{x \in \Omega} r_x p_x$$

is called expected value of random variable r.

Random variable

A (measurable) function $\Omega \to X$ from sample space to some set X. It maps "random experiment outcomes" to some mathematically tractable objects.

e.g. real numbers, but can be other things too.

$\mathcal{R}(\Omega)$ – the set of all random variables on $(\Omega, \mathcal{E}, P)$.

Expected value

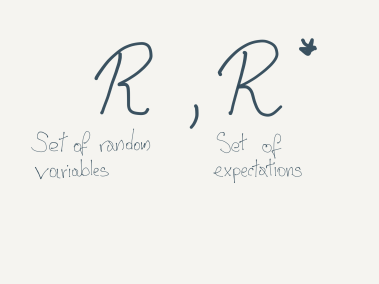
If $r \in \mathcal{R}(\Omega)$ is a random variable, then

$$\mathbb{E}(r) = \int_\Omega r(x)\, dp(x) = \int_\Omega r(x) p(x)\, dx = \sum_{x \in \Omega} r_x p_x$$

is called expected value of random variable r.

Denote by $\mathcal{R}^*(\Omega)$ set of all *expectations*.

Getting rid of sample space



$(\mathcal{R}, \mathcal{R}^*)$ the pair of (set of random variables, set of expectations) is totally sufficient to formulate all probability theory.

Note that this does not mention Ω!

Expressing idea in pseudo-Haskell

Random variable type

```haskell
type RVar a = Ω -> a -- random variable of type a
```

Random variable type

```haskell
type RVar a = Ω -> a -- random variable of type a

instance Functor RVar where
    fmap :: (a -> b) -> RVar a -> RVar b
    fmap f rv = f . rv
```

Random variable type

```haskell
type RVar a = Ω -> a -- random variable of type a

instance Functor RVar where
    fmap :: (a -> b) -> RVar a -> RVar b
    fmap f rv = f . rv
```

Let's just substitute for `RVar`:

```haskell
fmap :: (a -> b) -> (Ω -> a) -> (Ω -> b)
```

Random variable type

```haskell
type RVar a = Ω -> a -- random variable of type a

instance Functor RVar where
    fmap :: (a -> b) -> RVar a -> RVar b
    fmap f rv = f . rv
```

Let's just substitute for `RVar`:

```haskell
fmap :: (a -> b) -> (Ω -> a) -> (Ω -> b)
```

We can easily verify functor laws:

```haskell
fmap id rx = id . rx = rx
```

Random variable type

```haskell
type RVar a = Ω -> a -- random variable of type a

instance Functor RVar where
    fmap :: (a -> b) -> RVar a -> RVar b
    fmap f rv = f . rv
```

Let's just substitute for `RVar`:

```haskell
fmap :: (a -> b) -> (Ω -> a) -> (Ω -> b)
```

We can easily verify functor laws:

```haskell
fmap id rx = id . rx = rx

fmap (g . f) rx = (g . f) . rx
(fmap g . fmap f) rx = fmap g (fmap f rx) = fmap g (f . rx) = g . (f . rx)
```

Random variable type

It's a monad too, in pseudo-code:

```
type RVar a = Ω -> a | (Ω,Ω) -> a | (Ω,Ω,Ω) -> a | ...

instance Monad RVar where
    return :: a -> RVar a
    --        a -> (Ω -> a)
    return x = const x
```

Random variable type

It's a monad too, in pseudo-code:

```
type RVar a = Ω -> a | (Ω,Ω) -> a | (Ω,Ω,Ω) -> a | ...

instance Monad RVar where
    return :: a -> RVar a
    --        a -> (Ω -> a)
    return x = const x

    (>>=) :: RVar a -> (a -> RVar b)   -> RVar b
    --     (Ω -> a) -> (a -> (Ω -> b)) -> ((Ω,Ω) -> b)
    rv >>= f = f . rv
    -- because Ω -> Ω -> b is isomorphic to (Ω,Ω) -> b
```

Random variable type

It's a monad too, in pseudo-code:

```
type RVar a = Ω -> a | (Ω,Ω) -> a | (Ω,Ω,Ω) -> a | ...

instance Monad RVar where
    return :: a -> RVar a
    --        a -> (Ω -> a)
    return x = const x

    (>>=) :: RVar a -> (a -> RVar b)   -> RVar b
    --     (Ω -> a) -> (a -> (Ω -> b)) -> ((Ω,Ω) -> b)
    rv >>= f = f . rv
    -- because Ω -> Ω -> b is isomorphic to (Ω,Ω) -> b
```

We got rid of Ω!

The `random-fu` package

Basic blocks

Random variable is a basic type:

```
data RVar a  -- random variable of type a

instance Monad RVar
```

and *probability distribution* is a type-class:

```
class Distribution d t where
    rvar :: d t -> RVar t
```

It helps to construct random variables, e.g.:

```
stdUniform :: Distribution StdUniform a => RVar a
```

standard uniform distribution of type `a`;

e.g. uniform on $[0, 1]$ interval for `Double`, random int for `Int`, random `Bool`

```
normal :: Distribution Normal a => a -> a -> RVar a
```

normal (Gaussian) random variable of type `a` with given mean and standard deviation;

defined only for `Double` and `Float`.

Sampling random variables

At some point, we want to get the “random value”. Random variable can be *sampled:*

types are simplified for clarity

```
sample     :: MonadRandom  m   => RVar a -> m a
sampleFrom :: RandomSource m s => s -> RVar a -> m a
```

`MonadRandom` is a monad that has default source of entropy. Instance is defined for `IO`.

`RandomSource` is a specific source of entropy in monad `m`. Instances exist for standard haskell packages providing randomness (`mersenne-random-pure64`, `mwc-random`, `random`).

A bit of live coding

Stochastic process

Random walk

- we have set of states, represented by some type, e.g.
  ```
  type State = (Int, Int)  -- rectangular grid
  ```

Random walk

- we have set of states, represented by some type, e.g.
  ```
  type State = (Int, Int)  -- rectangular grid
  ```
- our "system" jumps from one state to the other, randomly:
  ```
  jump :: State -> RVar State
  ```
  Next state depend on current state.

Random walk

- we have set of states, represented by some type, e.g.
  `type State = (Int, Int)  -- rectangular grid`
- our “system” jumps from one state to the other, randomly:
  `jump :: State -> RVar State`

  Next state depend on current state.
- we iterate process k-times starting from some point `s0`:

```
run :: Int -> (State -> RVar State) -> State -> RVar State
run n t s0 = t s0 >>= process (n-1) t
```

Let’s put it together.

Applications

Classification rule mining

- `Rule` is a predicate on a database `Database` (it selects rows)

Classification rule mining

- `Rule` is a predicate on a database `Database` (it selects rows)
- we have function that transforms `Database` instance to rule selecting that particular instance.

```
toRule :: Database -> Int -> Rule
```

Classification rule mining

- `Rule` is a predicate on a database `Database` (it selects rows)
- we have function that transforms `Database` instance to rule selecting that particular instance.
  ```
  toRule :: Database -> Int -> Rule
  ```
- rules can be combined:
  ```
  (\/) :: Rule -> Rule -> Rule
  ```

Classification rule mining

- `Rule` is a predicate on a database `Database` (it selects rows)
- we have function that transforms `Database` instance to rule selecting that particular instance.
  ```
  toRule :: Database -> Int -> Rule
  ```
- rules can be combined:
  ```
  (\/) :: Rule -> Rule -> Rule
  ```
- and scored:
  ```
  score :: Database -> Rule -> Database
  ```

Classification rule mining

- `Rule` is a predicate on a database `Database` (it selects rows)
- we have function that transforms `Database` instance to rule selecting that particular instance.
  ```
  toRule :: Database -> Int -> Rule
  ```
- rules can be combined:
  ```
  (\/) :: Rule -> Rule -> Rule
  ```
- and scored:
  ```
  score :: Database -> Rule -> Database
  ```

Algorithm

We iteratively build result, starting with $r_0 = \perp$ being a null rule. Build a pool of rules pool by transforming each Database instance with positive target into rule.

1. take out randomly rule q from the pool
2. combine it with previous rule $q' = q \vee r_{i-1}$
3. if the score is smaller than given threshold then $r_i = r_{i-1}$ otherwise $r_i = q'$

Algorithm

We iteratively build result, starting with $r_0 = \bot$ being a null rule. Build a pool of rules pool by transforming each Database instance with positive target into rule.

1. take out randomly rule q from the pool
2. combine it with previous rule $q' = q \vee r_{i-1}$
3. if the score is smaller than given threshold then $r_i = r_{i-1}$ otherwise $r_i = q'$

```
pickFromSet :: Set a -> RVar (a, Set a)

step :: Database -> Double -> (Rule, Set Rule) -> RVar (Rule, Set Rule)
step db thr (r, pool) = do
  (q, pool') <- pickFormSet pool
  return $ (update thr r q, pool')
```

Classification rule mining – implementation

Algorithm

We iteratively build result, starting with $r_0 = \bot$ being a null rule. Build a pool of rules pool by transforming each Database instance with positive target into rule.

1. take out randomly rule q from the pool
2. combine it with previous rule $q' = q \vee r_{i-1}$
3. if the score is smaller than given threshold then $r_i = r_{i-1}$ otherwise $r_i = q'$

```
pickFromSet :: Set a -> RVar (a, Set a)

step :: Database -> Double -> (Rule, Set Rule) -> RVar (Rule, Set Rule)
step db thr (r, pool) = do
  (q, pool') <- pickFormSet pool
  return $ (update thr r q, pool')

update :: Double -> Rule -> Rule -> Rule
update thr r q | score q' >= thr = q'
               | otherwise       = r
  where
    q' = r \/ q
```

Classification rule mining – implementation

Algorithm

We iteratively build result, starting with $r_0 = \bot$ being a null rule. Build a pool of rules pool by transforming each Database instance with positive target into rule.

1. take out randomly rule q from the pool
2. combine it with previous rule $q' = q \vee r_{i-1}$
3. if the score is smaller than given threshold then $r_i = r_{i-1}$ otherwise $r_i = q'$

```
pickFromSet :: Set a -> RVar (a, Set a)

step :: Database -> Double -> (Rule, Set Rule) -> RVar (Rule, Set Rule)
step db thr (r, pool) = do
  (q, pool') <- pickFormSet pool
  return $ (update thr r q, pool')

update :: Double -> Rule -> Rule -> Rule
update thr r q | score q' >= thr = q'
               | otherwise       = r
  where
    q' = r \/ q

mine = Database -> Double -> RVar Rule
mine db thr = runUntil emptyPool (step db thr) (pool, nullRule)
  where
    pool = toSet . fmap toRule . filter positiveTarget $ db
```

Comparision to older implementation (Python + C)

- much simpler code base
  - algorithm in Haskell version is well isolated in about 50 lines of code;
  - previous implementation mix core algorithm with manipulation and has total 1715 lines of code.
- much faster execution time (about $100\times$ of real time speed-up)
- much easier development (took around 1 week)

Benchmark MWC vs random-fu

Task: compute mean value 10000 random double drawn from uniform distribution on unit interval

	non-optimized [ms]	optimized [ms]
random-fu	1.800	0.890
MWC	7.112	0.391

MWC code is improves very much when optimization is enabled. This suggest that both libs may on on-par in less trivial programs.