

# MODELING PROBABILITY IN HASKELL PROGRAMMING LANGUAGE

ONDREJ ŠUCH

ABSTRACT. In this note we provide examples how to model work with probability spaces in Haskell programming language.

The textbook *Concrete mathematics* [CoMa] is used in many courses of mathematics for computer scientists. In the chapter 8, there is an introduction to various notions of probability theory. In this article we undertake to provide an implementation of software to explicitly model these notions in Haskell [Hask] programming language.

## 1. FINITE PROBABILITY SPACES

A finite probability space consists of a finite set  $\Omega$  of elementary events and a real valued probability function  $p : \Omega \rightarrow \mathbf{R}$  satisfying two axioms:

$$(1) \quad p(\omega) \geq 0, \quad \text{for elementary events } \omega \text{ in } \Omega,$$

$$(2) \quad \sum_{\omega \in \Omega} p(\omega) = 1.$$

A natural data structure to model such a space is a collection of pairs, where the first component of a pair is an elementary event, and the second is the value of the probability function. Thus our model will be a finite subset of  $\Omega' := \Omega \times \mathbf{R}$  consisting of elements  $(\omega, p(\omega))$ . It will be convenient to work with the function  $pr : \Omega' \rightarrow \mathbf{R}$  defined by

$$pr(o, p) = p$$

**1.1. Modeling the space.** To begin with, we choose Haskell's list data structure as the underlying collection. Then a fair and an unfair coin can be described simply as:

```
fairCoin = [ ("Head", 1/2), ("Tail", 1/2) ]
unfairCoin = [ ("Head", 3/7), ("Tail", 4/7) ]
```

We can also define the probability function for an element of the space by extracting the second component of a pair by Haskell's library function `snd`:

```
pr x = snd x
```

Having defined the spaces and the probability function it is now straightforward to implement a function to check the axioms.

---

*Date:* January 30, 2009.

*Key words and phrases.* Haskell, probability.

```
isProbSpace space = firstAxiomHolds && secondAxiomHolds
  where
    firstAxiomHolds = all (\ x -> pr x >= 0) space
    secondAxiomHolds = sum (map pr space) == 1
```

It is striking how little needs to be added to the mathematical ideas in (1) and (2). Even the unnamed boolean function (*lambda expression*) defined in the definition of `firstAxiomHolds` is quite comprehensible.

**1.2. Real vs. rational numbers.** Once one starts using `isProbSpace` to verify various probability spaces, the reality of floating point arithmetic bites. Because of rounding, some probability spaces are erroneously flagged as not satisfying the second axiom.

The nice thing is that Haskell comes with libraries that implement rational numbers. Thus exact rational numbers are used for probabilities in the following definitions of dice:

```
import Data.Ratio
```

```
sides = [1,2,3,4,5,6]
uniformSpace events = zip events [p, p ..]
                      where p = 1 % (length events)
die0 = uniformSpace sides
die1 = zip sides [1%4, 1%8, 1%8, 1%8, 1%8, 1%4]
```

The space `die0` represents the fair die with all probabilities equal to 1/6, whereas the space `die1` represents an unfair die with variable probabilities. In both cases, the function `isProbSpace` now correctly reports that they satisfy the axioms for a probability space.

The astonishing thing for a C++ programmer may be that the function `isProbSpace` works without any change! The *type inferring* feature of Haskell adjusts the definition of `isProbSpace` by itself.

**1.3. Cartesian products.** For a mathematician a natural way to construct new probability spaces is to form Cartesian products. To express this way of thinking directly one can use Haskell's *list comprehensions*.

```
event x = fst x
cartesianProduct space1 space2 =
  [ ( (event e1, event e2), (pr e1) * (pr e2) ) |
    e1 <- space1, e2 <- space2 ]
```

Having this definition, we can define probability spaces corresponding to various combinations of throws by the fair and unfair dice from the previous section.

```
dice00 = cartesianProduct die0 die0
dice01 = cartesianProduct die0 die1
dice11 = cartesianProduct die1 die1
```

It almost goes without saying that the function `isProbSpace` will correctly identify all of them as satisfying axioms of a probability space.

## 2. RANDOM VARIABLES

A random variable is just a function from the set of elementary events to some other set. Here are some examples of random variables:

- is the number of a die even? (boolean random variable)
- the sum of numbers thrown on two dice (integer random variable)
- the quotient of numbers thrown on two dice of different colors (rational random variable)

**2.1. Push-forward of probability space.** Given a random variable  $Y : \Omega \rightarrow \Pi$ , it is possible to construct a probability space on  $\Pi$  by defining for elements  $\pi$  in  $\Pi$

$$p(\pi) = \sum_{\omega:Y(\omega)=\pi} p(\omega).$$

To construct this probability space, it is necessary to coalesce elements of the list `map (\ x -> (Y x, pr x)) space` with identical values of `Y x`. We can use `FiniteMap` library of Haskell to accomplish this.

```
image f space = fmToList (foldl addToImage (listToFM []) space)
  where
    addToImage map x = addToFM map (fx) (previousProb + (pr x))
      where
        fx = f (event x)
        previousProb = case lookupFM map fx of
          Nothing -> 0;
          Just y -> y;
```

This is an example of an accumulation process on a list using `foldl` function. A finite map, initially created empty by `(listToFM [])`, is used to hold a dictionary of events and their probabilities.

**2.2. Cumulative distribution.** While the folding function `foldl` and its variants are introduced to Haskell learners at an early stage, their cousin `scanl` which returns the partial compositions is less used. Computing cumulative distribution is actually the perfect application of this function.

```
cdf f space = zip (map event sortedSpace) probabilities
  where
    sortedSpace = sortWith event (image f space)
    probabilities = tail (scanl (\x y -> x + (pr y)) 0 sortedSpace)
```

**2.3. Mean, median, mode.** Three common ways to find the “average” of a random variable are to compute the mean, median and mode. All of these functions are easily implemented in Haskell:

```
mean f space = sum (map (\x -> f(event x) * pr(x)) space)

mode f space = map event (filter (\x -> pr x == maxProb) range)
  where
    range = image f space
    maxProb = maximum (map pr range)
```

```

median f space = if (snd left) == 1/2
                  then [event left , event (head (tail cum))]
                  else [event left , event left]

where
  cum = dropWhile (\x -> snd x < 1/2) (cdf f space)
  left = head cum

```

Let us look at the type of these functions, as inferred by Haskell.

```

*Pr> :info mean
mean :: (Num b) => (a -> b) -> [(a, b)] -> b
      — Defined at pr.hs:103:0-3
*Pr> :info mode
mode :: (Ord b, Num b, Ord key) => (a -> key) -> [(a, b)] -> [key]
      — Defined at pr.hs:105:0-3
*Pr> :info median
median ::
  (Fractional b, Ord b, Ord a1) => (a -> a1) -> [(a, b)] -> [a1]
      — Defined at pr.hs:110:0-5

```

We can see that Haskell compiler automaticall identified to what random variables various functions apply. In order to compute mean the random variable has to have values that belong to **Num** class, which means it can be computed for instance for integer, or real random variables. Median can be computed with random variables, whose values belong to **Ord** class, that is they can be compared<sup>1</sup>

**2.4. Independent variables.** To check that two random variables **f1** and **f2** are independent one can define a function as follows:

```

areIndependent f1 f2 space = all equalProb range
where
  range1 = listToFM (image f1 space)
  range2 = listToFM (image f2 space)
  range = image (\x -> (f1 x, f2 x)) space
  equalProb w = prXY == prX * prY
where
  prXY = snd w
  prX = case (lookupFM range1 (fst (event w)) ) of Just z -> z
  prY = case (lookupFM range2 (snd (event w)) ) of Just z -> z

```

One subtle point to notice is the potential performance gain. Because of Haskell's *laziness*, the round-trip conversions between `FiniteMap` and list representations used for `range1` and `range2` can be avoided.

---

<sup>1</sup>Let us remark that even though `mode` can be computed for arbitrary random variables, Haskell correctly flagged our implementation as requiring the random variable taking values in **Ord** class. This is because we used `FiniteMap` for `image` function.

## 3. SAMPLING

Sampling of events on probability spaces is important for two reasons.

Firstly, from the point of view of probability theory, it gives students a hands-on experience with probabilities. A student can use sampling to numerically refute a homework computation based on a numerical error. It can help to show counterintuitive examples such as Penney's game [WaPe]. In this game there are two players A, and B, who bet on winning sequences while throwing a coin. The player A wins if the pattern (head, head, tail) is thrown first, the player B wins if the pattern (head, tail, tail) occurs first. Even though the patterns have equal probability characteristics, the player A is twice as likely to win as the player B!

Secondly, from the point of view of programming in a pure functional language, sampling is an important example of monadic actions. The function to generate random number, or random events are not purely functional, because they depend on the "state of the world", more precisely on the current state of the random generator.

**3.1. EventGenerator data structure.** One way to generate a sequence of random events on a probability space is to maintain the array of cumulative probabilities and lookup random values in the interval  $[0, 1]$  in this array.

To maintain precomputed state during event generation we define a data structure, which parallels template types in languages like C++, C# or Java:

```
data EventGenerator eventType probType = EventGenerator
  {
    arrEvents :: Array Int eventType ,
    arrCumPrs :: Array Int probType
  }
```

To construct an instance of this type, we implement a function to generate it from a probability space:

```
fromSpace space = EventGenerator {
  arrEvents = aEvents ,
  arrCumPrs = aCumPrs}

where
  lEvents = fst (unzip space)
  lCumPrs = tail (scanl (\ x y -> x + (pr y)) 0 space)
  bds = (0::Int , length (lEvents) - 1)
  (aEvents , aCumPrs) = (listArray bds lEvents ,
    listArray bds lCumPrs)
```

We must implement a function to lookup the values in the array of cumulative probabilities (an equivalent of C function `bsearch`). It can be implemented as follows using *guards*:

```
findArray arr dice = findArray' (bounds arr)
where
  findArray' bds | a == b      = a
                 | mid >= dice = findArray' (a , midIndex)
                 | midIndex > a = findArray' (midIndex , b)
```

```

    | otherwise = b
where (a,b) = bds
        midIndex = div (a + b) 2
        mid = unsafeAt arr midIndex

```

This function can be used once a random value `rVal` from interval  $[0, 1]$  is thrown via utility function

```

getEvent gen rVal = unsafeAt (arrEvents gen) idx
where idx = findArray (arrCumPrs gen) rVal

```

**3.2. Generating actions.** *Actions*, unlike pure functions, return results that depend on the current “state of the world”. They can also have side effects and thus alter the “state of the world”. A simple generation of a random value is one of the simplest examples of an action. It effects the hidden state of the global random number generator. Therefore the next random value is different from the one currently obtained. Multiple actions need to be sequenced with `do` keyword that specifies the order in which they need to be executed.

To generate a random event from a given `EventGenerator` structure is now a matter of the simple composition of two actions in *IO monad*

```

genEvent gen = do rVal <- getStdRandom (randomR (0.0, 1.0))
                return (getEvent gen rVal)

```

To implement sampling in Penney’s game, let us define a function using `Maybe` data type:

```

penneyGame step threeThrows
  | threeThrows == ["Head", "Head", "Tail"] = Just "A_won"
  | threeThrows == ["Head", "Tail", "Tail"] = Just "B_won"
  | otherwise = Nothing

```

or to implement sampling of how many times to take to throw a head we can use function

```

isHead step event = if event == ["Tail"] then Just step
                   else Nothing

```

The general action to carry out sampling using either `isHead` or `penneyGame` function is implemented as follows:

```

seqSample (f,n) space = do initials <- forM [1..n] genAction
                        testStage n initials

where
  gen = fromSpace space
  genAction _ = genEvent gen
  testStage step events = case (f step events) of
    Just z -> return (z);
    Nothing -> do
      newEvent <- genEvent gen
      testStage (step + 1) ((tail events) ++ [newEvent])

```

## 4. CONCLUSION

We have demonstrated how to very concisely<sup>2</sup> implement basic probability functions in Haskell. In the course of doing so we have demonstrated the following features of Haskell language, some of which are not found in various languages currently used in university education:

- list comprehensions
- exact rational numbers
- lambda expressions
- laziness
- guards
- type inferring
- **Maybe** data type
- monadic actions

Haskell, unlike say R language, is not specifically geared towards programming probability and statistics functions. It is thus rather remarkable that standard libraries sufficed to carry out our goals.

Future directions to investigate include adding GUI for students' interaction or parallelizing sampling.

We would like to conclude with our gratitude to the organizers of Winter School on Probability '09 in Remata, which inspired this paper, as well as to Marián Grendár, Ivan Brodenec and Tobey Kenney for their help and suggestions during preparation of this paper.

## REFERENCES

- [CoMa] Graham Ronald L., Knuth Donald E., Patashnik Oren, *Concrete mathematics: a foundation for computer science*, 1989, Addison-Wesley
- [Hask] Haskell programming language, <http://www.haskell.org/>
- [WaPe] Penney Walter, "Problem 95: Penney-Ante," *Journal of Recreational Mathematics* 7 (1974), 321

INSTITUTE OF MATHEMATICS AND COMPUTER SCIENCE, ĎUMBIERSKA 1, 974 01 BANSKÁ BYSTRICA, SLOVAK REPUBLIC,

*E-mail address:* [ondrejs@savbb.sk](mailto:ondrejs@savbb.sk)

---

<sup>2</sup>To the code shown it remains to just add import directives to the following libraries: **GHC.Arr**, **FiniteMap**, **System.Random**, **Control.Monad**, **Util**