JFP **16** (1): 21–34, 2006. © 2005 Cambridge University Press
doi:10.1017/S0956796805005721 Printed in the United Kingdom

FUNCTIONAL PEARLS

Probabilistic functional programming in Haskell

MARTIN ERWIG and STEVE KOLLMANSBERGER
School of Electrical Engineering and Computer Science,
Oregon State University, Corvallis, OR 97331, USA
(*e-mail:* [erwig, kollmast]@eecs.oregonstate.edu)

1 Introduction

At the heart of functional programming rests the principle of referential transparency, which in particular means that a function f applied to a value x always yields one and the same value $y = f(x)$. This principle seems to be violated when contemplating the use of functions to describe probabilistic events, such as rolling a die: It is not clear at all what exactly the outcome will be, and neither is it guaranteed that the same value will be produced repeatedly. However, these two seemingly incompatible notions can be reconciled if probabilistic values are encapsulated in a data type.

In this paper, we will demonstrate such an approach by describing a probabilistic functional programming (PFP) library for Haskell. We will show that the proposed approach not only facilitates probabilistic programming in functional languages, but in particular can lead to very concise programs and simulations. In particular, a major advantage of our system is that simulations can be specified independently from their method of execution. That is, we can either fully simulate or randomize any simulation without altering the code which defines it. In the following we will present the definitions of most functions, but also leave out some details for the sake of brevity. These details should be obvious enough to be filled in easily by the reader. In any case, all function definitions can be found in the distribution of the library, which is freely available at `eecs.oregonstate.edu/~erwig/pfp/`.

The probabilistic functional programming approach is based on a data type for representing *distributions*. A distribution represents the outcome of a probabilistic event as a collection of all possible values, tagged with their likelihood.

```
newtype Probability = P Float
newtype Dist a = D {unD :: [(a,Probability)]}
```

This representation is shown here just for illustration; it is completely hidden from the users of the library by means of functions which construct and operate on distributions. In particular, all functions for building distribution values enforce the constraint that the sum of all probabilities for any non-empty distribution is 1. In this way, any `Dist` value represents the complete sample space of some probabilistic event or “experiment”.

Distributions can represent events, such as the roll of a die or the flip of a coin. We can construct these distributions from lists of values using *spread* functions, that is, functions of the following type.

```
type Spread a = [a] -> Dist a
```

The library defines spread functions for various well-known probability distributions, such as `uniform` or `normal`, and also a function `enum` that allows users to attach specific probabilities to values. With `uniform` we can define, for example, the outcome of die rolls.

```
die = uniform [1..6]
```

Probabilities can be extracted from distributions through the function `??` that is parameterized by an *event*, which is represented as a predicate on values in the distribution.

```
type Event a = a -> Bool

(??) :: Event a -> Dist a -> Probability
(??) p = P . sum . map snd . filter (p . fst) . unD
```

There are principally two ways to combine distributions: If the distributions are independent, we can obtain the desired result by forming all possible combinations of values while multiplying their corresponding probabilities. For efficiency reasons, we can perform normalization (aggregation of multiple occurrences of a value). The normalization function is mentioned later.

```
joinWith :: (a -> b -> c) -> Dist a -> Dist b -> Dist c
joinWith f (D d) (D d') = D [(f x y,p*q) | (x,p) <- d, (y,q) <- d']

prod :: Dist a -> Dist b -> Dist (a,b)
prod = joinWith (,)
```

Examples of combined independent events are rolling a number of dice. The function `certainly` constructs a distribution of one element with probability 1.

```
dice :: Dist [Int]
dice 0 = certainly []
dice n = joinWith (:) die (dice (n-1))
```

On the other hand, if the second event depends on the first, it must be represented by a function that accepts values of the distribution produced by the first event. In other words, whereas the first event can be represented by a `Dist a` value, the second event should be a function of type `a -> Dist b`. This dependent event combination is nothing other than the bind operation when we regard `Dist` as a monad.

```
instance Monad Dist where
  return x     = D [(x,1)]
  (D d) >>= f  = D [(y,q*p) | (x,p) <- d, (y,q) <- unD (f x)]
  fail         = D []
```

The functions `return` and `fail` can be used to describe outcomes that are certain or impossible, respectively. We also use the synonyms `certainly` and `impossible` for these two operations. We will also need monadic composition of two functions and a list of functions.

```
(>@>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
f >@> g = (>>= g) . f

sequ :: Monad m => [a -> m a] -> a -> m a
sequ = foldl (>@>) return
```

We have defined `Dist` also as an instance of `MonadPlus`, but this definition is not important for this paper.

The observation that probability distributions form a monad is not new (Giry, 1981). However, previous work was mainly concerned with extending languages by offering probabilistic expressions as primitives and defining suitable semantics (Jones & Plotkin, 1989; Morgan *et al.*, 1996; Ramsey & Pfeffer, 2002; Park *et al.*, 2004). The focus of those works is on identifying semantics to support particular aspects, such as the efficient evaluation of expectation queries in (Ramsey & Pfeffer, 2002) by using a monad of probability measures or covering continuous distributions in addition to discrete ones by using sampling functions as a semantics basis (Park *et al.*, 2004) (and sacrificing the ability to express expectation queries). However, we are not aware of any work that is concerned with the design of a probability and simulation library based on this concept.

Having defined distributions as monads allows us to define functions to repeatedly select elements from a collection without putting them back, which causes later selections to be dependent on earlier ones. First, we define two functions that, in addition to the selected element, also return the collection without that element.

```
selectOne :: Eq a => [a] -> Dist (a,[a])
selectOne c = uniform [(v,List.delete v c) | v <- c]

selectMany :: Eq a => Int -> [a] -> Dist ([a],[a])
selectMany 0 c = return ([],c)
selectMany n c = do (x,c1)  <- selectOne c
                    (xs,c2) <- selectMany (n-1) c1
                    return (x:xs,c2)
```

Since in most applications the elements that remain in the collection are not of interest, it is helpful to provide derived functions that simply project onto the values of the distributions. The implementation reveals that `Dist` is also a functor. We will also use the function name `mapD` to refer to `fmap` to emphasize that the mapping is across distributions.

```
instance Functor Dist where
  fmap f (D d) = D [(f x,p) | (x,p) <- d]

mapD :: (a -> b) -> Dist a -> Dist b
mapD = fmap
```

With `mapD` we can now define the functions for repeatedly selecting elements from a collection. Note that the function `fst` is used in `select` because `selectMany` returns a tuple containing the list of selected elements and the list of remaining (unselected) elements. We wish to discard the latter. We reverse the returned list because the elements retrieved in `selectMany` are successively cons'ed onto the result list, which causes the first selected element to be the last in that list.

```
select :: Eq a => Int -> [a] -> Dist [a]
select n = mapD (reverse . fst) . selectMany n
```

With this initial set of functions we can already approach many problems found in textbooks on probability and statistics and solve them by defining and applying probabilistic functions. For example, what is the probability of getting at least 2 sixes when throwing 4 dice? We can compute the answer through the following expression.

```
> ((>=2) . length . filter (==6)) ?? dice 4
 13.2%
```

Another class of frequently found problems is exemplified by "What is the probability of drawing a red, green, and blue marble (in this order) from a jar containing two red, two green, and one blue marble without putting them back?". With an enumeration type for marbles containing the constructors `R`, `G`, and `B`, we can compute the answer as follows.

```
> (==[R,G,B]) ?? select 3 [R,R,G,G,B]
  6.7%
```

Many more examples are contained in the library distribution.

A final concept that is employed in many examples is the notion of a *probabilistic function*, that is, a function that maps values into distributions. For example, the second argument of the bind operation is such a probabilistic function. Since it turns out that in many cases the argument and the result type are the same, we also introduce the derived notion of a *transition* that is a probabilistic function on just one type.

```
type Trans a = a -> Dist a
```

A common operation for a transition is to apply it to a distribution, which is already provided through the bind operation.

In the next two sections, we illustrate the use of basic library functions with two examples to demonstrate the high-level declarative style of probabilistic functional programming. In section 4 we will show how randomization fits into the described approach and, in particular, how it allows the approximation of distributions to cope with exponential space growth. In section 5 we describe how to deal with traces of probabilistic computations. Conclusions in section 6 complete this paper.

2 The Monty Hall problem

In the Monty Hall problem, a game show contestant is presented with three doors, one of which hides a prize. The player chooses one of the doors, and then the host opens another door which does not have the prize behind it. The player then has the option of staying with the door they have chosen or switching to the other closed door. This problem is also discussed in Hehner (2004) and Morgan *et al.* (1996). When presented with this problem, most people will assume that switching makes no difference – since the host has opened a door without the prize, it leaves a 50/50 chance of the remaining two doors.

However, statistical analysis has shown that the player doubles their chance of winning if they switch doors. How can this be? We can use our library to determine if this analysis is correct, and how.

A simple approach is to first consider that of the three doors, only one is the winning door. Thus, the player's initial pick has a one third chance of being the winning door.

```
data Outcome = Win | Lose

firstChoice :: Dist Outcome
firstChoice = uniform [Win,Lose,Lose]
```

If the player has chosen a winning door, and then switches, they will lose. However, if they initially chose a losing door, the host only has one choice for a door to open: the other losing door. Thus, if they switch, they win. This process can be captured by a transition on outcomes.

```
switch :: Trans Outcome
switch Win  = certainly Lose
switch Lose = certainly Win
```

We can analyze the probabilities of winning by comparing `firstChoice` and applying the transition `switch` to `firstChoice`.

```
*MontyHall> firstChoice
Lose  66.7%
 Win  33.3%

*MontyHall> firstChoice >>= switch
 Win  66.7%
Lose  33.3%
```

Therefore, not switching gives the obvious one third chance of winning, while switching gives a two thirds chance of winning.

We can also model the game in more detail, replicating each step with the precise rules that accompany them. We first construct the structure of the simulation from the bottom up. We start with three doors.

```
data Door = A | B | C

doors :: [Door]
doors = [A .. C]
```

Next we create a data structure to represent the state of the game by having fields that indicate which door (`A`, `B`, or `C`) contains the prize, which is chosen, and which is opened.

```
data State = Doors {prize :: Door, chosen :: Door, opened :: Door}
```

Of course, these will not all be assigned at once, but in sequence. In the initial state, the prize has not yet been hidden, no door has been chosen, and no door is open. Since the state will be evaluated only after all fields are set, we can initialize all fields with `undefined`.

```
start :: State
start = Doors {prize=u,chosen=u,opened=u} where u=undefined
```

Now each step of the game can be modeled as a transition on `State`. First, the host will choose one of the doors at random to hide the prize behind.

```
hide :: Trans State
hide s = uniform [s{prize=d} | d <- doors]
```

A transition takes a value of some type and produces a distribution of that type. In this case, the transition `hide` takes a `State` and produces a uniform distribution of states – one state for each door the prize could be hidden behind. Next, the contestant will choose, again at random, one of the doors.

```
choose :: Trans State
choose s = uniform [s{chosen=d} | d <- doors]
```

Once the contestant has chosen a door, the host will then open a door that is not the one chosen by the contestant and is not hiding the prize. This is the first transition which depends on the value of `State` it receives by considering the value of `s` in the definition.

```
open :: Trans State
open s = uniform [s{opened=d} | d <- doors \\ [prize s,chosen s]]
```

Next the player can switch or stay with the door already chosen. Both strategies can be represented as transitions on `State`.

```
type Strategy = Trans State
```

Switching means chose a door that is currently not chosen and that has not been opened.

```
switch :: Strategy
switch s = uniform [s{chosen=d} | d <- doors\\ [chosen s,opened s]]
```

We can also create a strategy for stay, which would simply be to leave everything precisely as it already is.

```
stay :: Strategy
stay = certainlyT id
```

For constructing transitions which produce a distribution of only one element, we provide the function `certainlyT` which converts any function of type `a -> a` into a function of type `a -> Dist a`.

```
certainlyT :: (a -> a) -> Trans a
certainlyT f = certainly . f
```

Finally, we define an ordered list of transitions that represents the game: hiding the prize, choosing a door, opening a door, and then applying a strategy.

```
game :: Strategy -> Trans State
game s = sequ [hide,choose,open,s]
```

Recall that `sequ` implements the composition of a list of monadic functions, which are transitions in this example.

If, once all the transitions have been applied, the chosen door is the same as the prize door, the contestant wins.

```
result :: State -> Outcome
result s = if chosen s==prize s then Win else Lose
```

We can define a function `eval` that plays the game for a given strategy and computes the outcome for all possible resulting states.

```
eval :: Strategy -> Dist Outcome
eval s = mapD result (game s start)
```

Again, we can determine the value of both strategies by computing a distribution.

```
*MontyHall> eval stay
Lose  66.7%
 Win  33.3%

*MontyHall> eval switch
 Win  66.7%
Lose  33.3%
```

Note that the presented model can be easily extended to four (or more) doors. All we have to do is add a `D` constructor to the definition of `Door` and change `C` to `D` in the definition of `doors`. A third take on this example will be briefly presented in section 4.

3 An example from biology: tree growth

Many applications in biology are based on probabilistic modeling. In fact, the motivation for creating the PFP library results from a joint project with the Center for Gene Research and Biotechnology at Oregon State University in which we have developed a computational model to explain the development of microRNA genes (Allen *et al.*, 2005).

Consider the simple example of tree growth. Assume a tree can grow between one and five feet in height every year. Also assume that it is possible, although less likely, that a tree could fall down in a storm or be hit by lightning, which we assume would kill it standing. How can this be represented using probabilistic functions?

We can create a data type to represent the state of the tree and, if applicable, its height.

```
type Height = Int
data Tree = Alive Height | Hit Height | Fallen
```

We can then construct a transition function for each state that the tree could be in.

```
grow :: Trans Tree
grow (Alive h) = normal [Alive k | k <- [h+1..h+5]]
```

When the tree is alive, it grows between 1 and 5 feet every year. We distribute these values on a normal curve to make the extreme values less likely.

```
hit :: Trans Tree
hit (Alive h) = certainly (Hit h)

fall :: Trans Tree
fall _ = certainly Fallen
```

When the tree is hit, it retains its height, but when fallen, the height is discarded. We can combine these three transitions into one transition that probabilistically selects which action should happen to a live tree.

```
evolve :: Trans Tree
evolve t@(Alive _) = unfoldT (enum [0.9,0.04,0.06] [grow,hit,fall]) t
evolve t           = certainly t
```

Here we use the function `enum` to create a custom spread with the given probabilities. We apply this spread to the list of transitions (`grow`, `hit`, and `fall`) which creates a distribution of transitions. The function `unfoldT` converts a distribution of transitions into a regular transition.

```
unfoldT :: Dist (Trans a) -> Trans a
```

This transition is then applied to `t`, the state of the tree, to produce a final distribution for that year. With an initial value, such as `seed = Alive 0`, we can now run simulations of the tree model.

Table 1. *Comparing the maximum heap size (in kilobytes) for fully simulated and randomized tree growth simulations*

Generations	Fully simulated	Randomized (500 runs)
5	700	650
6	4000	800
7	19000	800

To find out the situation after several generations, it is convenient to have a combinator that can iterate a transition a given number of times or while a certain condition holds. Three such combinators are collected in the class `Iterate`, which allows the overloading of the iterators for transitions and randomized changes (see next section).

```
class Iterate c where
  (*.)  :: Int -> (a -> c a) -> (a -> c a)
  while :: (a -> Bool) -> (a -> c a) -> (a -> c a)
  until :: (a -> Bool) -> (a -> c a) -> (a -> c a)
  until p = while (not . p)
```

For example, to compute the distribution of possible tree values after `n` years, we can define the following function.

```
tree :: Int -> Tree -> Dist Tree
tree n = n *. evolve
```

For large values of `n`, computing complete distributions is computationally infeasible. In such cases, randomization of values from distributions provides a way to approximate the final distribution with varying degrees of precision. We will discuss this randomization in the next section.

Since the distribution spreads out into many different values quickly, we do not show an example run here.

4 Randomization

The need for randomization arises quickly when an iterated transition creates a new set of values for each value currently in the distribution, thus creating an exponential space explosion. We provide functions to transform a regular transition into a *randomized change*, which select only one result from the created distribution. By repeatedly applying such a randomized change to the same value, we can construct an arbitrarily good approximation of the exact probabilistic distribution. An example of the space explosion compared to the roughly constant space requirement in randomization is shown in Table 1 for the tree growth simulation given in the previous section.

Library users need not be concerned about randomization when they first design transitions. Instead, randomization can be performed later automatically by employing corresponding library functions. Likewise, tracing can be provided on request by the library without changes to the transition function itself. In this way, tracing and randomization are concepts that are orthogonal to the library's probabilistic modeling capabilities.

All randomized values live within the `R` monad, which is simply a synonym for `IO`. Elementary functions to support randomization are `pick`, which selects exactly one value from a distribution by randomly yielding one of the values according to the specified probabilities (performed by `selectP`), and `random`, which transforms a transition into a randomized change.

```
type R a = IO a
type RChange a = a -> R a

pick :: Dist a -> R a
pick d = Random.randomRIO (0,1) >>= return . selectP d

random :: Trans a -> RChange a
random t = pick . t
```

Randomly picking a value from a distribution or randomizing a transition is not an end in itself. Instead, the collection of values obtained by repeated application of randomized changes can be aggregated to yield an approximation of a distribution, represented by the type `RDist a`, representing a *randomized distribution*. Given a list of random values, we can first transform them into a list of values within the `R` monad. Then we can assign equal probabilities with `uniform` and group equal values by a function `norm` that also sums their probabilities.

```
type RDist a = R (Dist a)
type RTrans a = a -> RDist a

rDist :: Ord a => [R a] -> RDist a
rDist = fmap (norm . uniform) . sequence
```

With `rDist` we can implement a function `~.` that repeatedly applies a randomized change or a transition and derives a randomized distribution. The `Ord` constraint on `a` in the signature of `~.` is required because the instance definitions are based on `norm` (through `rDist`).[1]

[1] The function `norm` sorts the values in a distribution to achieve grouping for efficiency reasons. Since we have found that in most examples that we encountered defining an `Ord` instance is not more difficult than an `Eq` instance, we have preferred the more efficient over the more general definition.

```
class Sim c where
  (~.) :: Ord a => Int -> (a -> c a) -> RTrans a

instance Sim IO where
  (~.) n t = rDist . replicate n . t

instance Sim Dist where
  (~.) n = (~.) n . random
```

In particular, the latter instance definition allows us to simulate transitions in retrospect. In other words, we can define functions to compute full distributions and can later turn them into computations for randomized distributions without changing their definition. For example, the tree growth computation that is given by the function `tree` can be turned into a simulation that runs a randomized tree growth `k` times as follows.

```
simTree :: Int -> Int -> Tree -> RDist Tree
simTree k n = k ~. tree n
```

Similarly, for the Monty Hall problem we could randomly perform the trial many times instead of deterministically calculating the outcomes.

```
simEval :: Int -> Strategy -> RDist Outcome
simEval k s = mapD result `fmap` (k ~. game s) start
```

Since in many simulation examples it is required to simulate the `n`-fold repetition of a transition `k` times, we also introduce a combination of the functions `*.` and `~.` that performs both steps.

```
class Sim c where
  (~*.) :: Ord a => (Int,Int) -> (a -> c a) -> RTrans a

instance Sim IO where
  (~*.) (k,n) t = k ~. n *. t

instance Sim Dist where
  (~*.) x = (~*.) x . random
```

Note that `*.` is defined to bind stronger than the `~.` function. We can thus implement the tree simulation also directly based on `evolve`.

```
simTree k n = (k,n) ~*. evolve
```

Again, we do not have to mention random number generation anywhere in the model of the application.

5 Tracing

As simulation complexity increases, some computational aspects become difficult. If, for example, we wished to evaluate the growth of the tree at each year for one hundred years, it would be quite redundant to calculate it first for one year, then separately for two years, and again for three, and so on. Instead, we could calculate the growth for one hundred years and simply keep track of all intermediate results.

To facilitate tracing, we define types and functions to produce traces of probabilistic and randomized computations. For deterministic and probabilistic values we introduce the following types.

```
type Trace a  = [a]
type Space a  = Trace (Dist a)
type Walk a   = a -> Trace a
type Expand a = a -> Space a
```

A walk is a function that produces a trace, that is, a list of values. Continuing the idea of iteration described in the previous section, we define a function to generate walks, which is simply a bounded version of the predefined function `iterate`.

```
walk :: Int -> Change a -> Walk a
walk n f = take n . iterate f
```

Note that the type `Change a` is simply a synonym for `a -> a`, introduced for completeness and symmetry (see `RChange a` in the previous section).

While a walk produces a trace, iteration of a transition yields a list of distributions, which represents the explored probability space. We use the symbol `*..` to represent the trace-producing iteration. The definition is based on the function `>>:`, which prepends the result of a transition to a space, and `singleton`, which maps `x` into `[x]`.

```
(*..) :: Int -> Trans a -> Expand a
0 *.. _ = singleton . certainly
1 *.. t = singleton . t
n *.. t = t >>: (n-1) *.. t

(>>:) :: Trans a -> Expand a -> Expand a
f >>: g = \x -> let ds@(D d:_)=g x in
                    D [(z,p*q) | (y,p) <- d,(z,q)<- unD (f y)]:ds
```

The potential space problem of plain iterations is even worse in trace-producing iterations. Therefore, we also define randomized versions of the types and iterators.

```
type RTrace a  = R (Trace a)
type RSpace a  = R (Space a)
type RWalk a   = a -> RTrace a
type RExpand a = a -> RSpace a
```

The function `rWalk` iterates a random change to create a random walk, which can produce a random trace.

```
rWalk :: Int -> RChange a -> RWalk a
```

The definition is similar to that of `*..`, but not identical, because the result types are structurally different: While `Dist` is nested within `Trace` in the result type of `*..`, `R` is wrapped around `Trace` in the result type of `rWalk`. This structural difference in the result types is also the reason why we cannot overload the notation for these two functions.

Similar to `~.` we can now implement a function `~..` that simulates the repeated application of a randomized change or transition and derives a randomized space, that is, a randomized sequence of distributions that approximate the exact distributions. Since `~..` is overloaded like `~.` for transitions and random changes, it can reside in the same class `Sim`. The function `replicate k` produces a list containing `k` copies of the given argument. In the second instance definition, `mergeTraces` transposes a list of random lists into a randomized list of distributions, which represent an approximation of the explored probabilistic space.

```
class Sim c where
  (~..) :: Ord a => (Int,Int) -> (a -> c a) -> RExpand a

instance Sim IO where
  (~..) (k,n) t = mergeTraces . replicate k . rWalk n t

instance Sim Dist where
  (~..) x = (~..) x . random
```

Note that the first argument of `~..` is a pair of integers representing the number of simulation runs *and* the number of repeated application of the argument function. The latter is required to build the correct number of elements in the trace unlike for `~.` where only the final result matters.

Applied to the tree-growth example we can now define functions for computing an exact and approximated history of the probabilistic tree space as follows.

```
hist :: Int -> Tree -> Space Tree
hist n = n *.. evolve

simHist :: Int -> Int -> Tree -> RSpace Tree
simHist k n = (k,n) ~.. evolve
```

6 Conclusion

We have illustrated an approach to express probabilistic programs in Haskell. The described ideas are implemented as a collection of Haskell modules that are combined into a *probabilistic functional programming library*. In addition to the examples described here, the PFP library contains modules defining functions for queueing simulations, bayesian networks, predator/prey simulations, dice rolling, card playing, etc. Moreover, the library provides a visualization module to produce plots and figures that can be rendered by the `R` package (Maindonald & Braun, 2003).

Probabilistic functional programming can be employed as a constructive approach to teach (or study!) statistics, as shown in the first two sections. Moreover, the PFP library provides abstractions that allow the high-level modeling of probabilistic scientific models and their execution or simulation. We have illustrated this aspect here with a toy example, but we have successfully applied the library to investigate a real scientific biological problem (Allen *et al*., 2005). In particular, the high-level abstractions allowed us to quickly change model aspects, in many cases immediately during discussions with biologists about the model.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments on an earlier version of this paper.

References

Allen, E., Carrington, J., Erwig, M., Kasschau, K. and Kollmansberger, S. (2005) Computational Modeling of microRNA Formation and Target Differentiation in Plants. In preparation.

Giry, M. (1981) A categorical approach to probability theory. In: Banaschewski, B. (editor), *Categorical Aspects of Topology and Analysis*. Lecture Notes in Mathematics 915, pp. 68–85.

Hehner, Eric C. R. (2004) Probabilistic predicative programming. *7th Int. Conf. on Mathematics of Program Construction: LNCS 3125*, pp. 169–185.

Jones, C. and Plotkin, G. D. (1989). A probabilistic powerdomain of evaluations. *4th IEEE Symp. on Logic in Computer Science*, pp. 186–195.

Maindonald, J. and Braun, J. (2003) *Data Analysis and Graphics Using R*. Cambridge University Press.

Morgan, C., McIver, A. and Seidel, K. (1996) Probabilistic predicate transformers. *ACM Trans. Program. Lang. Syst.* **18**(3), 325–353.

Park, S., Pfenning, F. and Thrun, S. (2004) A probabilistic language based upon sampling functions. *32nd Symp. on Principles of Programming Languages*, pp. 171–182.

Ramsey, N. and Pfeffer, A. (2002) Stochastic lambda calculus and monads of probability distributions. *29nd Symp. on Principles of Programming Languages*, pp. 154–165.

https://doi.org/10.1017/S0956796805005721 Published online by Cambridge University Press