040: State

When working through Chapter 6 (and these exercises) it is important to have a picture of the emerging story. This chapter, with the progression of exercises, shows you how we can arrive at an elegant abstraction in library design. We are starting with imperative generator of random numbers (no exercises about this), then we make it pure (the `RNG` state). Then again we observe that using it is tedious so, we start to see the random number generators as state transformers (`Rand`) and then we remark that the main functions manipulating them can just as well manipulate any state transformers, not only random number generators (we arrive at the `State` abstraction). To understand this story well, you will likely have to re-read the book in parallel to solving the exercises.

All exercises are to be solved by extending the file `State.scala`. It contains several modules inside, to avoid name clashes between different abstractions. Once done, please hand in the completed `State.scala` file.

Exercise 1. Write a function that uses `RNG.nextInt` to generate a random integer between 0 and `Int.maxValue`. Make sure to handle the corner case when `nextInt` returns `Int.MinValue`, which doesn't have a non-negative counterpart.[1]

```
def nonNegativeInt(rng: RNG): (Int, RNG)
```

Implement the function in the companion object of the `RNG` trait, file `State.scala`.

Exercise 2. Write a function to generate a `Double` between `0` and `1`, not including `1`. Note: You can use `Int.MaxValue` to obtain the maximum positive integer value, and you can use `x.toDouble` to convert an `x: Int` to a `Double`.

```
def double(rng: RNG): (Double, RNG)
```

Write this function in the companion object of the `RNG` trait.[2] Ponder for a moment: is the distribution generated by your function uniform?

Exercise 3. Write functions to generate an (Int, Double) pair and a (Double, Int) pair, where integers are non-negative. You should be able to reuse the functions you've already written.

```
def intDouble (rng: RNG)
def doubleInt (rng: RNG)
```

Write all of these in the `RNG` companion object.[3]

Exercise 4. Write a function to generate a list of random integers.[4]

```
def ints(count: Int)(rng: RNG)
```

If you have not done it yet, please add explicit return type annotations to all above functions, and see that your solutions still type check. Notice that all functions written so far have the same format: `f[A] : RNG =>(A,RNG)`, except that in the last case this type has been curried with one additional parameter. This motivates the generalization of the interface from now on, see Section 6.4 in the text book for the `Rand[A]` type. We shall develop an API for this type, like we did for `Option`. The API will allow us computing with random values, without explicitly carrying the generator state around.

[1] Exercise 6.1 [Chiusano, Bjarnason 2014]
[2] Exercise 6.2 [Chiusano, Bjarnason 2014]
[3] Exercise 6.3 [Chiusano, Bjarnason 2014]
[4] Exercise 6.4 [Chiusano, Bjarnason 2014]

Exercise 5. Use map to reimplement `double` in a more elegant way. See exercise 2 above.

Observe how for `Option` we used the higher order API to avoid using pattern matching, and how here we use it to avoid being explicit about the state (and also to avoid decomposing, a.k.a. pattern matching, the results of random generators into value and new state).[5]

Exercise 6. Write the implementation of `map2` based on the following signature. This function takes two actions, `ra` and `rb`, and a function `f` for combining their results, and returns a new action that combines them:[6]

```
def map2[A,B,C](ra: Rand[A], rb: Rand[B])(f: (A, B) =>C): Rand[C]
```

Exercise 7. Implement `sequence` for combining a `List` of transitions into a single transition. Use it to reimplement the `ints` function you wrote before. For the latter, you can use the standard library function `List.fill(n)(x)` to make a list with `x` repeated `n` times.[7]

```
def sequence[A](fs: List[Rand[A]]): Rand[List[A]]
```

Exercise 8. Implement `flatMap`, and then use it to implement `nonNegativeLessThan`.[8]

```
def flatMap[A,B](f: Rand[A])(g: A =>Rand[B]): Rand[B]
```

Note: for `Option` we used `map` to compose a partial computation with a total computation. In here we used `map` to compose a random generator with a deterministic function. Similarly for `flatMap`. Function, `Option.flatMap` was used to compose two partial computations. On `Rand` the `flatMap` function is used to compose two random generators.

Exercise 9. Now we shall observe that everything we have done so far can just as well be done for other states, than `RNG`. Read beginning of Chapter 6.5, before solving this exercise.

Generalize the functions `unit`, `map`, `map2`, `flatMap`, and `sequence`. Add them as methods on the `State` case class where possible. Otherwise you should put them in a `State` companion object.[9]

Exercise 10. We now connect the `State` and `Streams`. Recall from basics of computer science that automata and traces are intimately related: each automaton generates a language of traces. In our implementation automata are implemented using `State` and `Streams` can be used to represent traces.

Implement a function `state2stream` that given a state object and an initial state, produces a stream of values generated by this `State` object (this automaton).

This composition provides an alternative to using `sequence` suggested above.

Exercise 11. Use `state2stream` to generate a lazy stream of integer numbers. Finally, obtain a finite list of 10 random values from this stream.

Notice, how concise is the expression to obtain 10 random values from a generator using streams. This is because all our abstractions compose very well.

[5] Exercise 6.5 [Chiusano, Bjarnason 2014]
[6] Exercise 6.6 [Chiusano, Bjarnason 2014]
[7] Exercise 6.7 [Chiusano, Bjarnason 2014]
[8] Exercise 6.8 [Chiusano, Bjarnason 2014]
[9] Exercise 6.10 [Chiusano, Bjarnason 2014]