Prop: Design of a Property-Based Testing Library

The purpose is to practice API design on the case of the property-based testing framework. The exercise may appear difficult. Note that this exercise is not about testing (done last week), but about designing and developing a testing framework.

Warning: We are developing our own PBT library, which is very similar but not identical to scalacheck. Not all experience from last week will transfer! Exercise 3 is an exception, where we do use ScalaCheck.

The file `State.scala` is the one developed in Chapter 6. We are not changing it, just using.

Hand in `Exercises.scala`.

Exercise 1. Recall the API of Chapter 6. `RNG` is the type of (R)andom (N)umber (G)enerators. Create a new `Simple` random number generator of type `RNG` and seed it with `42`.

Exercise 2. Get a pseudo random number (`x`) of the generator defined in the first exercise. Get the next random number and bind it to `y`.

Exercise 3. Write two ScalaCheck (!) properties that specify a function that finds the **minimum** of a `List[Int]`. You need to figure out what the two properties are yourself, but there is a hint: the two corresponding properties for `maximum` were included in the slides on testing, a week before.[1]

```
def minimum: List[Int] =>Int
```

The exercise file has a context set up so that ScalaCheck is available in this, and only in this, exercise. Use the generator of non-empty lists provided. Each property is put in a function taking an implementation of `minimum` to be tested.

Exercise 4. Implement `&&` (conjunction) as an infix method of `Prop`, for the implementation of `Prop` provided in the file. This method should create a new `Prop` object which checks to be true, if both combined methods are true. Do not call `check` during an application of your `&&`, but only when someone calls `check` on the conjoined property.[2]

Exercise 5. Implement a test case generator `Gen.choose`. It should generate integers in the range `start` to `stopExclusive`. Assume that `start` and `stopExclusive` are non-negative numbers. [3]

```
def choose(start: Int, stopExclusive: Int): Gen[Int]
```

Hint: Before solving the exercise study the type `Gen` in `Gen.scala`. Then, think how to convert a random integer to a random integer in a range. Then recall that we are already using generators that are wrapped in `State` and the state has a `map` function. The tests for this exercise will not pass until you have solved Exercise 9 (`flatMap`).

Exercise 6. Implement test case generators `unit` (always generates a constant value given to it in a parameter), `boolean` (generates randomly true, false), and `double` (generates random doubles).[4]

---

[1] Exercise 8.2 [Pilquist, Chiusano, Bjarnason, 2022]
[2] Exercise 8.3 [Pilquist, Chiusano, Bjarnason, 2022]
[3] Exercise 8.4 [Pilquist, Chiusano, Bjarnason, 2022]
[4] Exercise 8.5 with some changes [Pilquist, Chiusano, Bjarnason, 2022]

Hints: (i) The `State` trait already had `unit` implemented. (ii) How do you convert a random integer number to a random Boolean? (iii) Recall from two weeks ago that we already implemented a random number generator for doubles, which can be wrapped here.

Exercise 7. Implement an extension method `listOfN` for `Gen[A]` that given an integer number `n` returns a list of length `n` containing `A` elements, generated by the generator the method is called on.[5]

Hint: Recall that the standard library has the following useful function (in the `List` companion object): `def fill[A](n: Int)(elem: =>A): List[A]`

It is possible to implement a solution without it, but the result is ugly—you need to replicate the behavior of `fill` inside `listOfN`. You can use `fill` to create a list of generators. To turn the list of generators into a generator of lists, use the `State`'s `sequence` method. This can be used to execute a series of consecutive generations, passing the `RNG` state around.

Exercise 8. Explain in English why `listOfN` was implemented as an extension method of `Gen[A]`, not as a usual method.

Exercise 9. Implement `flatMap` for generators. Recall that `flatMap` allows to run another generator on the result of the present one (`this`). Note that in the type below the parameter `A` is implicitly bound, as this is also an extension method of `Gen[A]`:[6]

```
def flatMap[B] (f: A =>Gen[B]): Gen[B]
```

Hint: Recall that `Gen` is essentially a wrapped `State` of special kind. We already have a method `flatMap` for states, which allows to chain execution of automata. The simplest (and probably the best) solution is to delegate to that method.

Exercise 10. Use `flatMap` to implement a more dynamic version of `listOfN`:

```
def listOf(size: Gen[Int]): Gen[List[A]]
```

This version doesn't generate lists of a fixed size, but uses a generator of integers to pick the size first.[7]

Exercise 11. Implement `union`, for combining two generators of the same type into one, by pulling values from each generator with equal chance.[8]

```
def union[A](g1: Gen[A], g2: Gen[A]): Gen[A]
```

Hint: We already have a generator that emulates tossing a coin (which one is it?). Use `flatMap`.

Exercise 12. This exercise explores the concept of type classes (and 'givens'), which is important not only for testing and generators, but is a general extension mechanism used broadly in functional programming.

Reimplement functions `listOfN[A]` and `ListOf[A]` as top-level functions, not methods of `Gen`. The main challenge is to devise a type for the functions that uses an instance of a type class `Gen[A]` (the evidence that `A` is Generatable) to create instances of `A`. The body of the functions can actually just delegate to solutions of exercises 7 and 10. We would like the following calls to compile, if

[5] Exercise 8.5 [Pilquist, Chiusano, Bjarnason, 2022]
[6] Exercise 8.6 [Pilquist, Chiusano, Bjarnason, 2022]
[7] Exercise 8.6, second part [Pilquist, Chiusano, Bjarnason, 2022]
[8] Exercise 8.7 [Pilquist, Chiusano, Bjarnason, 2022]

instances of `Gen[Double]` and `Gen[Int]` are available:

```
Gen.listOfN[Double] (5)
Gen.listOf[Double]
```

Note that the ScalaCheck framework, used last week, separates `Gen` and `Arbitrary` for ergonomic reasons, to allow you to control better what is `given`. Only arbitraries are `given` with ScalaCheck. In this exercise, we avoided creating the additional `Arbitrary` type, and exposed `Gen` directly as a (given) type class.

Exercise 13. Recall that `Prop` is defined as:

```
opaque type Prop = (TestCases, RNG) => Result
```

`Prop` is a function that given some test cases and a random seed will produce a test result. Implement `Prop[A].&&` and `Prop[A].||` for composing `Prop` values. The former should succeed only if both composed properties (`this` and `that`) succeed; the latter should fail only if both composed properties fail. Recall that Exercise 4 was similar but for another representation of `Prop`.[9]

```
def && (p: Prop): Prop
def || (p: Prop): Prop
```

Hint: You can check whether a result is a failure by calling the `isFalsified` method on a `Result` value.

Exercise 14. (Context: Section 8.2) Implement a helper function for converting `Gen` to `SGen`. You can add this as an extension method on `Gen`. This function should just ignore the size altogether, so we have a 'broken' `Gen` but this way we can use our normal `Gen`s as `SGen`s.[10]

```
def unsized: SGen[A]
```

Exercise 15. Implement a list combinator that doesn't accept an explicit size. It should return an `SGen` instead of a `Gen`. The implementation should generate lists of the requested size.[11]

Exercise 16. Repeat Exercise 3 but now use our own testing framework to write the two tests.

In this exercise, the (ScalaCheck) test suite calls the tests you have written using our framework. Messages from our framework are marked, and printed in blue, to reduce confusion. Notice that some of these tests fail on purpose—when we test a testing framework we want to run both successful and failing tests. Your grade is only decided based on the messages from ScalaCheck (so the usual green/red/white messages).

[9] Exercise 8.9, first part [Pilquist, Chiusano, Bjarnason, 2022]
[10] Exercise 8.10 [Pilquist, Chiusano, Bjarnason, 2022]
[11] Exercise 8.12 [Pilquist, Chiusano, Bjarnason, 2022]