090: Design Patterns for Computation: Monoid, Foldable, Functor, Monad

This week we are learning the following skills:

- Using higher kinded types, and exploiting generic programming for advanced constructs
- Reusing computations, regardless of the object they operate on—this is a kind of 'super-generic' programming.

If you cannot complete all exercises in reasonable time, then skip some on Monoids, solve all for Foldables and Functors, and skip some for Monads. This way you try some in each of the sections. Hand in: `Monoid.scala`, `Monad.scala`, `MonoidSpec.scala`, `FunctorSpec.scala`, and `MonadSpec.scala`, 5 files (**not zipped**).

Monoids

Exercise 1. Give `Monoid` instances for integer addition, multiplication, and for Boolean operators.[1]

```
1 val intAddition: Monoid[Int]
2 val intMultiplication: Monoid[Int]
3 val booleanOr: Monoid[Boolean]
4 val booleanAnd: Monoid[Boolean]
```

This and the following exercises are best solved in the `Monoid` module (the companion object of the `Monoid` trait) in `src/main/scala/fpinscala/monoids/Monoid.scala`.

Exercise 2. Give a `Monoid` instance for combing `Option` values.[2] The composition operator should return its left argument if its not `None`, otherwise it should return the right argument.

```
def optionMonoid[A]: Monoid[Option[A]]
```

Exercise 3. A function having the same argument and return type is called an endofunction. Write a monoid for endofunctions:[3] `def endoMonoid[A]: Monoid[A =>A]`

Exercise 4. The file `src/test/scala/fpinscala/monoids/MonoidSpec.scala` formalizes the monoid laws as scalacheck properties. Make sure that you understand how it is done. Use `sbt test` command in the project root directory to run these tests.

A type constraint `[A :Arbitrary]` means that the type `A` has to implement the `Arbitrary` trait. This means that scalacheck will be able to generate random instances of it. See scalacheck's user guide for basic introduction to using it, if you find the laws cryptic (see https://github.com/rickynils/scalacheck/blob/master/doc/UserGuide.md).

Use this law formulation to test our other monoids implemented above (`intAddition`, `intMultiplication`, `booleanOr`, `booleanAnd`, and `optionMonoid`). Ensure that all tests pass. Solve this in `src/test/scala/fpinscala/monoids/MonoidSpec.scala`.[4]

Exercise 5. Write scalacheck tests that test whether a function is a homomorphism between two sets. Use them to test that `String` and `List[Char]` are isomorphic. (See Section 10 in the textbook about monoid homomorphisms). A string can be translated to a list of characters using the `toList` method. The `List.mkString` method with default arguments (no arguments) does the opposite conversion.

[1] Exercise 10.1 [Chiusano, Bjarnason 2014]
[2] Exercise 10.2 [Chiusano, Bjarnason 2014]
[3] Exercise 10.3 [Chiusano, Bjarnason 2014]
[4] Exercise 10.4 was the inspiration for this task [Chiusano, Bjarnason 2014]

Exercise 6. Now the implementation of monoid laws from the previous exercise to show that the two `Boolean` monoids from Exercise 1 above are isomorphic via the negation function (`!`).

The actual testing of monoid laws using scalacheck is not that interesting in itself. What is interesting is that you write property tests and you train generic programming. You should not reimplement the laws for the Boolean monoids, but the laws should have been made generic in the previous exercise. If not, generalize them to generic now.

Exercise 7. Implement a `productMonoid` that builds a monoid out of two monoids. Test it with scalacheck, say by composing an `Option[Int]` monoid with a `List[String]` monoid and running through our monoid laws. You should not need to write any new laws. Just reuse the existing ones.

```
def productMonoid[A,B] (ma: Monoid[A]) (mb: Monoid[B]): Monoid[(A,B)]
```

The monoid should be implemented in the `Monoid` companion objects, while the test should be placed in the `MonoidSpec.scala` file.

Foldables

Exercise 8. Implement `Foldable[List]`. You can place the implementation in the `Foldable` companion object in `Monoid.scala`.[5]

Exercise 9. Any `Foldable` structure can be turned into a `List`. Write this conversion in a generic way, as a member of the `Foldable` trait in `Monoid.scala`:[6]

```
def toList[A](fa: F[A]): List[A]
```

Notice, that here we are explicitly using the trait's mixed nature (`Foldable` has both abstract and concrete members, which allows to reuse `List` for different implementations of the interface).

Functors

Exercise 10. The `Functor` trait is implemented in `Monad.scala`. The companion object contains the `ListFunctor` instance. Implement an instance of `OptionFunctor` (in the companion object).

Exercise 11. Find the file `FunctorSpec.scala` and analyze how the map law is implemented there, in a way that it can be used for any functor instance. The type parameter `[F[_]]` is a type constructor in the same way as used in the functor definition. So the law holds for any type `A` and a type constructor `F[_]`. The second parameter of the law may seem mysterious. This is a, so called, implicit parameter. It states that when you use this method (`mapLaw`) there must exist an implicit conversion rule from `F[A]` instances to `ArbitraryF[A]` instances. This parameter is normally not provided explicitly at call site—the compiler finds a matching rule in the current name space.

Recall that scalacheck needs to know that `F[A]` is an instance (or can be made an instance) of `Arbitrary` in order to be able to generate random instances.

Below the law's definition we show how to use the law to test that `ListFunctor` is a functor (over integer lists). Note that indeed the implicit parameter is not provided. Scalacheck defines implicit conversions for `List[Int]` and these are matched automatically to `arb` at call place.

Now use the law to test that `OptionFunctor` over character strings is a functor (the one you made in the previous exercise).

[5] Exercise 10.12 [Chiusano, Bjarnason 2014]
[6] Exercise 10.15 [Chiusano, Bjarnason 2014]

The interesting aspects of this exercise are: (1) the use of type constructor parameters (higher-kinded types), (2) the use of implicit conversion to impose an interface on a generic type (it forces the user of the function to provide the conversion, if not implicitly available at call site). Reflect on these, to make sure that you understand how these function, so that you could use these constructs in your own projects. The concept of functor is somewhat less important.

Monads

Exercise 12. Write monad instances for `Option` and `List`. Remap standard library functions to the monad interface (or write from scratch). Put the implementations in the `Monad` companion object.[7]

Exercise 13. (Hard) Implement `sequence` as a method of the `Monad` trait. Express it in terms of `unit` and `map2`. Sequence takes a list of monads and merges them into one, which generates a list. Think about a monad as if it was a generator of values. The created monad will be a generator of lists of values—each entry in the list generated by one of the input monads.

```
def sequence[A] (lfa: List[F[A]]): F[List[A]]
```

Use sequence to run some examples. Sequence a list of instances of the list monad, and a list of instances of the option monad. Do you understand the results? Revisit the implementation of the `State.sequence`. What does it do?

This exercise provides a key intuition about the ubiquitous monad structure: it is a computational pattern for sequencing that is found in amazingly many contexts. Try to obtain an abstract understanding of this pattern. It will improve your abilities to see similar computations for reuse.[8]

Exercise 14. Implement `replicateM`, which replicates a monad instance `n` times into an instance of a list monad. This should be a method of the `Monad` trait.[9]

```
def replicateM[A](n: Int, ma: F[A]): F[List[A]]
```

Think how `replicateM` behaves for various choices of `F`. For example, how does it behave in the List monad? What about Option? Describe in your own words the general meaning of `replicateM`.

Exercise 15. The file `MonadSpec.scala` shows the monad laws implemented generically and tested on the option Monad. The design is very similar to the one for functors. Compare this with the description of laws in the book, and make sure you understand it. There is no new language or design concepts here, except perhaps that we need to use two implicit parameters, for identity.

Add test properties for list monad over integers, and a list monad over strings (in `MonadSpec.scala`).

Note that the laws are slightly restricted, they require that all composed monads are over the same type `A`. You can try to generalize them, so that multiple types can be supported. Since more types will be generated, you will need even more implicit parameters. This is a rather good advanced exercise in complex generic programming with higher kinded types.

Exercise 16. (It's getting abstract) Implement the Kleisli composition function compose (Sect. 11.4.2):[10]

```
def compose[A,B,C](f: A =>F[B], g: B =>F[C]): A =>F[C]
```

[7] Exercise 11.1 [Chiusano, Bjarnason 2014]
[8] Exercise 11.3 [Chiusano, Bjarnason 2014]
[9] Exercise 11.4–5 [Chiusano, Bjarnason 2014]
[10] Exercise 11.7 [Chiusano, Bjarnason 2014]