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

This week we are learning the following skills:

- Setting up your projects with scalacheck/scalatest property testing frameworks and sbt (useful for the project next week)
- Writing property tests for different structures
- Using higher kinded types, and exploiting generic programming for advanced constructs
- Abstract design: reusing computations, regardless of the object they operate on—this is a kind of 'super-generic' programming.

If you struggle with completing all exercises within reasonable time, then skip some on Monoids, solve all for Foldables and Functors, and skip some for Monads, so that you try to solve some in each of the sections.

Monoids

Exercise 1. (Exercise 10.1 in [Chiusano, Bjarnason 2015])

Give `Monoid` instances for integer addition and multiplication as well as the Boolean operators.

```
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. (Exercise 10.2 in [Chiusano, Bjarnason 2015])

Give a `Monoid` instance for combing `Option` values.

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

The composition operator should return its left argument if its not `None`, otherwise it should return the right argument.

Exercise 3. (Exercise 10.1 in [Chiusano, Bjarnason 2015])

A function having the same argument and return type is called an endofunction. Write a monoid for endofunctions.

```
def endoMonoid[A]: Monoid[A =>A]
```

Exercise 4. (inspired by Exercise 10.4 in [Chiusano, Bjarnason 2015])

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.

The test are run using `sbt test` command in the project root directory.

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 (`listMonoid`, `intAddition`,

`intMultiplication`, `booleanOr`, `booleanAnd`, and `optionMonoid`).

Make sure that all tests pass. The exercise is solved in `src/test/scala/fpinscala/monoids/-MonoidSpec.scala`. You can use this files for setting up your test environments in later projects. Also observe that scalacheck is added to the build environment in the sbt configuration file `build.sbt` in the project root.

Exercise 5. (Exercise 10.5 [Chiusano, Bjarnason 2015])

Implement `foldMap` (in the `Monoid` companion object).

```
def foldMap[A,B] (as: List[A], m: Monoid[B]) (f: A=>B): B
```

Exercise 6. (Exercise 10.7 [Chiusano, Bjarnason 2015])

Implement a `foldMap` for `IndexedSeq`. `IndexedSeq` is the interface for immutable data structures supporting efficient random access. It also has efficient `splitAt` and length methods. Your implementation `foldMap` should use the strategy of splitting the sequence in two, recursively processing each half, and then adding the answers together with the monoid.

```
def foldMapV[A,B](v: IndexedSeq[A], m: Monoid[B])(f: A =>B): B
```

Place your implementation in the `Monoid` companion object.

Exercise 7. Write scalacheck tests that test whether a function is a homomorphism between two sets. Then use it 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.

Exercise 8. 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 9. Implement a `productMonoid` that builds a monoid out of two monoids. Test it with scalacheck for instance 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 10. (Exercise 10.12, 10.14 [Chiusano, Bjarnason 2015]) Implement `Foldable[List]` and `Foldable[Option]`. You can place the implementations in the `Foldable` companion object in `Monoid.scala`.

Exercise 11. (easy, Exercise 10.15 [Chiusano, Bjarnason 2015])

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`:

```
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 12. The `Functor` trait is implemented in `Monad.scala`. The companion object contains the implementation of the `ListFunctor`. Implement an instance of `FunctorOption` (in the companion object).

Exercise 13. 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).

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 14. (Easy; Exercise 11.1 in [Chiusano, Bjarnason 2015])

Write monad instances for `Option`, `Stream` and `List`. You can remap standard library functions to the monad interface (or write your own from scratch). Place the implementations in the `Monad` companion object.

Exercise 15. (Hard; Exercise 11.3 [Chiusano, Bjarnason 2015])

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 synthesize an abstract understanding of this pattern. It will improve your abilities to see similar computations as candidates for reuse.

Exercise 16. (Easy; Exercise 11.4–5 [Chiusano, Bjarnason 2015])

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.

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

Think about how `replicateM` will behave 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 17. 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.

Then add test properties for list monad over integers, stream monad of integers, and stream monad over strings (in the `MonadSpec.scala` file).

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 18. (It's getting really abstract; Exercise 11.7 [Chiusano, Bjarnason 2015])

Implement the Kleisli composition function compose (see Section 11.4.2):

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

This and some of the following abstract exercise are very good brain teasers that develop your ability to think abstractly, and to solve abstract problems (similarly to algorithmic exercises).

Exercise 19. Find the monad laws stated using Kleisli's composition (they are in the book in sections 11.4.2–3). Implement this version of the law in `MonadSpec`. The law should be completely equivalent to the one presented in Exercise 17, so test it first on the same four examples. Then try something more crazy: for instance check associativity on Kleisli functions from `String` to `Int`, from `Int` to `Double`, and from `Double` to `Boolean` in the `Stream` monads.

Exercise 20. (Abstract and Hard; Exercise 11.8 [Chiusano, Bjarnason 2015])

Implement `flatMap` in terms of `compose` and `unit`. It seems that we've found another minimal set of monad combinators: `compose` and `unit`.