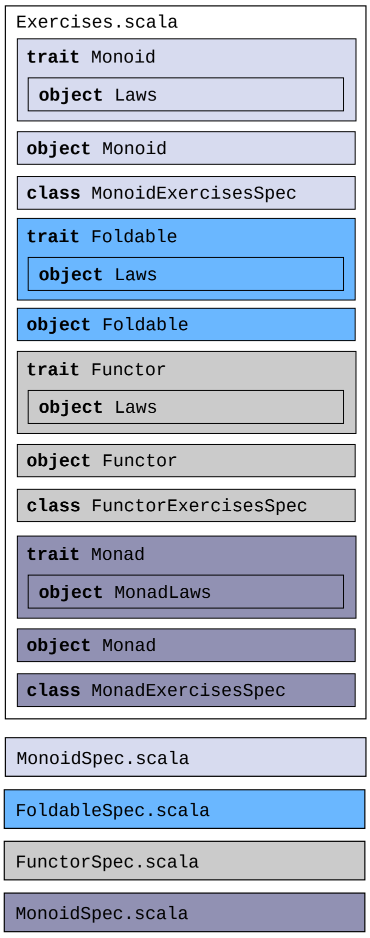
Design Patterns for Computation: Monoid, Foldable, Functor, Monad

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.
- Categorizing APIs into monoid, foldable, functor, and monad.



This is basically a generalization of what we have already seen "by-example" in the weeks discussing `Par`, `Prop` and `Parsers`.

In order to ensure that there is only one file to hand in, all exercises are placed in `Exercise.scala`. The layout of that file is summarized in the figure to the right. It contains implementations of monoids, foldables, functors, and monads. We would normally place them in separate files, but to make solving exercises more streamlined (less jumping around) and grading easier, they have been all placed in a single file. We even include those tests that you have to write as part of the homework in this file as well.

The tests have two parts for them. We implement the laws for each type class abstractly, in the nested object with laws. These laws cannot be executed directly, as they are polymorphic. They need to be instantiated for concrete argument type to test the type classes and their instances first. So writing tests this week mostly amounts to invoking the laws (sometimes the laws also need to be written). Finally we have some standalone spec files that contain pre-written tests to help you judge your progress.

Hand-in only `src/main/scala/adpro/Exercises.scala`. This is the only file you will be modifying this week.

Monoids

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

```
val intAddition: Monoid[Int]
val intMultiplication: Monoid[Int]
val booleanOr: Monoid[Boolean]
val booleanAnd: Monoid[Boolean]
```

We don't have (immediate) tests for this exercise. We develop them in exercise 4 below.

Exercise 2. Give a `Monoid` instance for combing `Option` values.[2]

a) We first define a monoid over `Monoid[A]` for any type `A`. The composition operator should return its left argument if it's not `None`, otherwise it should return the right argument.

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

b) Now consider another monoid for option. Assume that we have an instance of `Monoid` for `A` (so `A` is not anything, but we know that `A` is a monoid). Build a new monoid for `Option[A]` that always preserves `Some` over `None` when composing, and combines two `Some` values using the

[1]Exercise 10.1 [Chiusano, Bjarnason 2014]
[2]Exercise 10.2 [Chiusano, Bjarnason 2014]

operator of `Monoid[A]`. This allows us to promote any monoid `A` to a monoid in `Option[A]`:

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

You shall test these solutions in Exercise 4.

Exercise 3. A function having the same argument and return type is called an *endofunction*. Write a monoid instance for endofunctions:[3]

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

A natural monoid operator for endofunctions is function composition. The zero of the monoid of endofunctions will then be the identity function. Note that for any function `f`, if we compose it with identity, we still get `f`.

A test is provided for this exercise in `MonoidSpec.scala`.

Exercise 4. You will find a scalatest/scalacheck implementation of monoid laws in the top of the exercise file, in the object named `Monoid.Laws`. Understand how they are implemented (compare with the Laws in the monoid chapter of the textbook.)

Use these laws to test our other monoids implemented above: `intAddition`, `intMultiplication`, `booleanOr`, `booleanAnd`, `optionMonoid`, and `optionMonoidLift`. The last two are somewhat more tricky. We prepared a template for the solution in the same file in class `ExercisesMonoidSpec` below the `Monoid` object. Search below.

After the tests compile successfully, run them to tests solutions to Exercise 1 and 2.[4]

The test for Exercise 3 (used above to test `endoMonoid`) has an interesting twist: It needs to compare function values which is not supported in Scala (nor in any other mainstream programming language). We work around it by testing for equality of functions with a nested property test. Interested students are welcomed to study this test and compare with your solutions to Exercise 4.

Exercise 5. Implement `foldMap` (in the `Monoid` companion object).[5]

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

The function should convert the values on the list to `B`s and fold them using the monoid operator.

Exercise 6. Revisit Section 10.4 in the text book (the final subsection titled *Monoid Homomorphisms*) and understand the properties satisfied by a homomorphism and an isomorphism between monoids.

Write property-based tests that test whether a function is a homomorphism between two sets, and then combine them in the test of isomorphism. We write both laws in the object `Monoid.Laws` in the top part of the exercise file.

Note: With Scalatest we cannot use `andThen` or conjunction to combine homomorphisms into an isomorphism, like the book chapter does. This is an unfortunate consequence of Scalatest matchers being imperative (impure). You either just need to call them one after the other, just like in a regular imperative unit test, or you have to learn how to use the Scalacheck API from inside of Scalatest (considerably more work). Either way works for the purpose of this exercise, so just follow your interests.

We have no automatic test in this exercise but we will run and check it in the following one.

[3]Exercise 10.3 [Chiusano, Bjarnason 2014]
[4]Exercise 10.4 was the inspiration for this task [Chiusano, Bjarnason 2014]
[5]Exercise 10.5 [Chiusano, Bjarnason 2014]

Exercise 7. Recall that the text book (Section 10.1) has defined a monoid for Strings (with string concatenation and an empty string) and for Lists (with list concatenation and `Nil`). Intuitively, a character string and a list of characters are objects that carry the same information, they are similar, or more precisely isomorphic. We can always take a string and explode it into a list of individual characters, or take a list of characters and concatenate them to create a string. None of these operations appears to loose information.

Use the laws from the tests in the previous exercises to establish an isomorphism between `String` and `List[Char]` (or more precisely their monoids). Both monoids are already implemented in the top of the `Monoid` object. 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.

The exercise is to be solved inside `ExercisesMonoidSpec` — then you can run it with `sbt test`.

Exercise 8. Use the morphism laws from Exercise 6 to show that the two `Boolean` monoids from Exercise 1 above are isomorphic via the negation function (`!`).

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.

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

The monoid should be implemented in the `Monoid` companion object. The zero of the new monoid is a pair of zeros from the combined monoids. The operator of the new monoid, just applies the operators of the combined monoids point-wise, respectively to the left, and to the right parts of the combined elements.

We don't have (immediate) tests for this exercise, but we will use it in the next one.

Exercise 10. Test `productMonoid` using our monoid laws and Scalatest. You need to provide some concrete types for testing the product. We do not have generators for monoids, but we can compose some concrete types, for instance `Option[Int]` monoid with `List[String]` monoid. Run the resulting product monoid through our monoid laws. You should not need to write any new laws. Just reuse the existing ones.

The exercise is solved in `ExercisesMonoidSpec` below the `Monoid` object in the exercise file.

Foldables

Exercise 11. Implement `Foldable[List]`. Place the implementation in the `Foldable` companion object in `Exercises.scala`.[6]

Exercise 12. Any `Foldable` structure can be turned into a `List`. Write this conversion in a generic way, as a member of the `Foldable` trait in `Exercises.scala`:[7]

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

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

Functors

Exercise 13. The `Functor` trait is implemented in `Exercises.scala`. The companion object contains the `ListFunctor` instance. Implement an instance of `OptionFunctor` next to it.

We don't have (immediate) tests for this exercise. We develop them in the exercise below.

Exercise 14. Find the object `Functor.Laws` in `Exercises.scala` and analyze how the `map` law is implemented there, in a way that it can be used for any functor instance. The law holds for any type `A` and a type constructor `F[_]`, if we can generate arbitrary values of `F[A]`. Recall that Scalatest (Scalacheck) needs to know that there exists an instance of `Arbtirary` for `F[A]` in order to be able to generate random instances.

Below in the same file, in `FunctorExercisesSpec`, we show how to use the law to test that the `ListFunctor` is a functor (over integer lists). Note that indeed the implicit parameter is not provided. Scalatest/Scalacheck defines the necessary implicit instances. for `List[_]` and `Int` and these are matched automatically to `arb` at call site.

Use the law to test that `OptionFunctor` of Exercise 13 is a functor.

Monads

Exercise 15. 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.[8] We test these monad instances in the next exercise.

Exercise 16. The object `Monad.MonadLaws` in `Exercises.scala` shows the monad laws implemented generically. The design is very similar to the one for functors. Compare this with the description of laws in the book.

Add property tests for `optionMonad` and `listMonad` in the spec class `MonadExercisesSpec` in `Exercises.scala`

Exercise 17. 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]]
```

The file `MonadSpec.scala` has some tests for your implementation of `sequence` (`sbt test`). Try using sequence to run some examples in `sbt console`. 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 monad structure: A monad is a computational pattern for sequencing that is found in amazingly many contexts.[9]

Exercise 18. 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.[10]

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

```
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 19. (It's getting abstract)

Implement the Kleisli composition function compose (Sect. 11.4.2):[11]

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

The file `MonadSpec.scala` contains simple tests for this exercise.

[11]Exercise 11.7 [Chiusano, Bjarnason 2014]