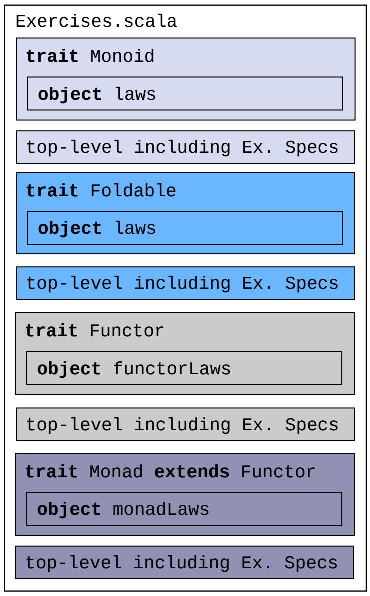
Design Patterns for Computation: Monoid, Foldable, Functor, Monad

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 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 `Exercises.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 (no jumping around) and grading easier, they have been all placed in a single file and extensions are used heavily.



The tests are split into two parts. 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 often amounts to invoking the laws.

Hand-in: `Exercises.scala`.

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 `Option`. The `empty` of this monoid will be `None`.[2]

a) We first define a monoid over `Monoid[Option[A]]` for any type `A`. The type `A` does not have to be a monoid itself. The composition operator `combine` should return its left argument if it's not `None`, otherwise it should return the right argument. It returns `None` in all other cases.

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

b) Now assume that the type `A` is a monoid itself, or more precisely we are `given` an an instance of `Monoid` for `A`. Build a new monoid for `Option[A]` that always preserves `Some` over `None` when combining like above, and combines two `Some` values using the operator of the given `Monoid[A]`. This allows us to promote any monoid `A` to a monoid in `Option[A]`, for free.

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

You shall test these solutions in Exercise 4.

[1] Exercise 10.1 [Pilquist, Chiusano, Bjarnason, 2022]
[2] Exercise 10.2 [Pilquist, Chiusano, Bjarnason, 2022]

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 the function composition. The `empty` will then be the identity function—for any function `f`, if we compose it with identity, we still get `f`.

The test for Exercise 3 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 4. You will find a scalacheck implementation of the monoid laws in the top of the exercise file, in the `Monoid` trait, under `laws`. Understand how they are implemented.

Use these laws to test our other monoids implemented above: `intAddition`, `intMultiplication`, `booleanOr`, `booleanAnd`, `optionMonoid`, and `optionMonoidLift`.[4]

Exercise 5. Implement `foldMap` (in the `Monoid` companion object). The function should convert the values on the list to `B`s and fold them using the monoid operator. [5]

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

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 above in the file. 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. 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.

[3]Exercise 10.3 [Pilquist, Chiusano, Bjarnason, 2022]
[4]Exercise 10.4 [Pilquist, Chiusano, Bjarnason, 2022]
[5]Exercise 10.5 [Pilquist, Chiusano, Bjarnason, 2022]

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

The `empty` of the new monoid is a pair of the `empty` values 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 will test this monoid constructor in the next exercise.

Exercise 10. Test `productMonoid` using our monoid laws and Scalacheck. You need to provide some concrete types for testing the product. We do not have generators of arbitrary monoids, so we cannot quantify over them in the test. Instead, 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.

Foldables

Exercise 11. Implement `Foldable[List]`.[6]

Exercise 12. Any `Foldable` structure can be turned into a `List`. Write this conversion in a generic way for any `F[_]: Foldable` and any `A`.[7]

We use the name `toListF`, not `toList` as there is a name conflict with something in our library conflict, and choosing a different name, will decrease the amount of debugging you will have to do.

Functors

Exercise 13. Implement an instance `Functor[Option]` of `Functor` for `Option`. We don't have (immediate) tests for this exercise. We develop them in the exercise below.

Exercise 14. Find the object `functorLaws` in the `Functor` trait (type class) 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]` and test for equality of `F[A]` values. Recall that Scalacheck needs to know that there exists an instance of `Arbtirary` for `F[A]` in order to be able to generate random instances. And we need a way to test equality to execute the property itself (the built-in equality `==` may not be suitable for all types, for instance functions).

Below we show how to use the law to test that the `ListFunctor` is a functor (over integer lists). Note that indeed the using parameter is not provided. Scalacheck defines the necessary given instances. for `List[_]` and `Int` and these are matched automatically to `arb`$*$ arguments at the 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 them from scratch).[8] We test these monad instances in the next exercise.

Exercise 16. The object `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. Use these laws to add property tests for `optionMonad` and `listMonad`.

[6] Exercise 10.12 [Pilquist, Chiusano, Bjarnason, 2022]
[7] Exercise 10.15 [Pilquist, Chiusano, Bjarnason, 2022]
[8] Exercise 11.1 [Pilquist, Chiusano, Bjarnason, 2022]

Exercise 17. Implement `sequence` as an extension method for lists of monadic values. 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. The classic type is:

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

but the exercise uses an extension, so that we do not have to jump around the file when adding solutions.

Now this single implementation of sequence does all what all our previous implementations did—this is truly mind-boggling! Use sequence to run some examples in the REPL. Sequence a list of instances of the list monad, and a list of instances of the option monad. We could also use it to sequence `Gens`, `Pars`, and `Parsers`, if we provided `Monad` instances for them. 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]

```
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]
```

[9]Exercise 11.3 [Pilquist, Chiusano, Bjarnason, 2022]
[10]Exercise 11.4–5 [Pilquist, Chiusano, Bjarnason, 2022]
[11]Exercise 11.7 [Pilquist, Chiusano, Bjarnason, 2022]