LIÈGE université

INFO0054-1
Programmation Fonctionnelle

Chapter 07: ADTs and Monoids

Christophe Debruyne

(c.debruyne@uliege.be)

References

- Chapter 10: Monoids.
  Paul Chiusano and Runar Bjarnason. Functional Programming in Scala, Manning Publications, 2015.

DISCLAIMER: Unlike languages such as Haskell, Scala does not provide built-in types for Foldable, Monoid, Functor, and Monad. But as they are abstract concepts representing objects that follow certain laws, we can easily represent those in Scala.

In the next few lessons, we will learn more about these abstract types. There are libraries for Scala that do provide these types. Examples include:

- **Scalaz https://github.com/scalaz/scalaz**
- **Cats https://typelevel.org/cats/**

Overview

- Algebraic Data Types
- Functional Design
- Monoids
- Monoid laws
- Foldable data structures
- Monoid composition

Part 01
Algebraic Data Types

Algebraic Data Types I

- "An algebraic data type [(ADT)] is a possibly recursive sum type of product types.
  - Each constructor "tags" a product type to separate it from others.
  - If there is only one constructor, the data type is a product type." [WikiPedia]

- Let's go back to our own implementation of a `List`.
- The ADT `List` is the sum of two products `Nil` and `Cons`.

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

- `List` is a sum type because a `List` is either (OR) a `Nil` or a `Cons`.
- `Cons` is a product type because it must contain a `head` (of any type) and (AND) a `tail` (of type `List`).
- `Nil` is product type because it must contain nothing.

Algebraic Data Types II

Much like the operators $+, -, \times$, ... we have while working with integers, ADTs allows us to work with all values a type can have by using $+$ (sum types) and $\times$ (product types). They are thus called algebraic because they have algebraic properties that are similar to integers.

Let's assume we are working with values of the type `Byte`, which are 8-bit signed values that range from -128 to 127 (256 values in total). We will use bytes to create points in a 2D and 3D space.

```
sealed trait Point
case class Point2D(x: Byte, y: Byte) extends Point
case class Point3D(x: Byte, y: Byte, z: Byte) extends Point
```

- `Point2D` can take on $256 \times 256 = 65536$ values.
- `Point3D` can take on $256 \times 256 \times 256 = 16777216$ values.
- `Point` can thus take on $65536 + 16777216 = 16842752$ values.

Some ADTs have an infinite set of values, but ADTs allow us to *reason* about types.

Algebra of Algebraic Data Types I

- We can work with the number of values of types.

- We can rewrite
  - `Point2D` can take on $256 \times 256 = 65536$ values.
  - `Point3D` can take on $256 \times 256 \times 256 = 16777216$ values.
  - `Point` can thus take on $65536 + 16777216 = 16842752$ values.
- As
  - $|\texttt{Point2D}| = 65536$
  - $|\texttt{Point3D}| = 16777216$
  - $|\texttt{Point}| = 16842752$

Algebra of Algebraic Data Types II

- Before we develop the algebra further, let's look at another example to develop an intuitive understanding.

```
sealed trait A
case class B(x: Boolean) extends A
case class C(x: Boolean) extends A
```

B has two values and C has two values; therefore, A can take on 4 values.

- $|B| = 2$
- $|C| = 2$
- $|A| = 4$

Algebra of Algebraic Data Types III

- In pure FP, all functions must return a value. Functions that do not return a value (`void` in some programming languages and `Unit` in Scala) are usually functions (or methods) with side-effects.

- In Scala, `Unit` is the return type of functions returning nothing. `Unit` has one singleton value that is `()`. `Unit` does provide us the identity value for $\times$:

$$|\tau \times Unit| = |\tau|$$

- Why? Let's take `Byte`, $Byte \times Unit$ returns us $\{(-128, ()), \ldots, (127, ())\}$. Adding the `Unit` type with a product does not provide us additional information and therefore the number of values remains the same.

Algebra of Algebraic Data Types IV

- Unlike other languages, Scala uses `Unit` to represent both the unit value (a type which only has one expression that has that type, namely `()`), and `void`, for which there are no expressions that evaluate to that type.
  - I.e., you cannot construct a `void`.

- We have no `void` in Scala (`Unit` plays the role of both). In programming language theory, `void` provides us the identity value for $+$:

$$|\tau + void| = |\tau|$$

- Why? As there are 0 expressions that have the type void, the number of values in $\tau + void$ is the same as τ.

Algebra of Algebraic Data Types V

- The sum type and product type operators follow the following algebraic rules:
  - $|\tau_1 \times \tau_2| = |\tau_1| \times |\tau_2|$
  - $|\tau_1 + \tau_2| = |\tau_1| + |\tau_2|$
- These algebraic rules give rise to the following algebraic properties
  - Commutativity
    - $|\tau_1 \times \tau_2| = |\tau_2 \times \tau_1|$
    - $|\tau_1 + \tau_2| = |\tau_2 + \tau_1|$
  - Associativity
    - $|(\tau_1 \times \tau_2) \times \tau_3)| = |\tau_1 \times (\tau_2 \times \tau_3)|$
    - $|(\tau_1 + \tau_2) + \tau_3)| = |\tau_1 + (\tau_2 + \tau_3)|$
  - Distributivity
    - $|\tau_1 \times (\tau_2 + \tau_3)| = |(\tau_1 \times \tau_2) + (\tau_1 \times \tau_3)|$

E.g., A `Point2D` can take on 65536 values, no matter the order of x and y.

Disclaimer

The algebra of ADTs is a theoretical discipline within programming languages research.

The goal of this chapter (and this course in general) is not to teach you about the algebra of ADTs. It suffices to know that in programming languages research, computer scientists study programming languages by formalizing aspects (syntax and semantics), give proofs to theorems (e.g., metatheory), …

By why are we covering some of these topics in a superficial manner?

General notions of what we touch upon, however, will be *applied* in this lesson.

For those who are interested in types, type systems and programming languages, I recommend (available on the Web): Pierce, B. C. (2002). Types and programming languages. MIT press.

Part 02
Functional Design

Functional Design I

- In the previous lessons, we have covered the principles of functional programming languages
  - No side effects, higher order functions, functional data structures, non-strict evaluation,…

- It turns out that many *problems* seem distinct but can be solved by a smaller set of *solutions*. They are merely combinations of certain structures that reappear. And when these structures reappear, there is an opportunity for code reuse.

- In the next two lessons, we will introduce new yet fundamental techniques in functional programming to avoid redundant code. Will try to detect common patterns and abstract them.

Functional Design II

- In the next two lessons, we will introduce new yet fundamental techniques in functional programming to avoid redundant code. Will try to detect common patterns and abstract them.

- That not only eliminates redundant (or duplicate) code, but also facilitates what is known as conceptual integration. Conceptual integration is the process of
  - identifying a common pattern (i.e., structure) among different solutions in different contexts,
  - uniting all these instances of that pattern under a single definition, and
  - giving that definition a name.

Algebras in FP III

- In FP, an algebra (not to be confused with the algebra of ADTs) is a set of functions operating over some data type(s) and a set of laws that specify relationships between these function.

- Such algebras are a way to create functional abstractions.

- We will cover some of these algebras in the next few lesson, with the first being monoids. You will notice that we have used monoids in the past!

Part 03
Monoids

Introduction I

- What do integer addition and string concatenation have in common?

- Integer addition
  - Has an identity element: 0
    - $3 + 0 = 3 = 0 + 3$
  - Is associative
    - $(3 + 1) + 5 = 5 + (3 + 1)$

- String concatenation
  - Has an identify element: "" (the empty string)
    - "foo" + "" = "foo" = "" + "foo"
  - Is associative
    - ("foo" + "bar") + "!" = "foo" + ("bar" + "!")

Introduction II

- What do integer addition and string concatenation have in common?

- Integer addition
  - Has an identity element: 0
    - $3 + 0 = 3 = 0 + 3$
  - Is associative
    - $(3 + 1) + 5 = 5 + (3 + 1)$

There is a common algebraic pattern, and that pattern abides by the same laws: an identify element and being associative.

They have very little in else common.

- String concatenation
  - Has an identity element: "" (the empty string)
    - "foo" + "" = "foo" = "" + "foo"
  - Is associative
    - ("foo" + "bar") + "!" = "foo" + ("bar" + "!")

Introduction III

This patterns appear in many places!

- Integer addition
- String concatenation
- Integer multiplication
  - Identity element is 1
    - $3 \times 1 = 3 = 1 \times 3$
  - Associative
    - $2 \times (3 \times 4) = (2 \times 3) \times 4$
- List concatenation
  - Identity element is `Nil`.
    - `List(1,3) ++ Nil = List(1,3) = Nil ++ List(1,3)`
  - Associative:
    - `List(0) ++ (List(1) ++ List(2)) = (List(0) ++ List(1)) ++ List(2)`

Monoids

- We can represent these patterns as monoids.

- A monoid consists of
  - Some type A.
  - An *associative* binary operation over that type A.
  - A value of type A that is the *identity* of the binary operation.

- The laws of associativity and identity are called the monoid laws.

- Monoids are simple purely algebraic structures as they are defined only by their algebra. In other words, other than abiding by the same laws, instances of the monoid interface may have very little in common.

Implementing some monoids

```
package monoids

trait Monoid[A]:
    def op(o1: A, o2: A): A
    val id: A


object Monoid:
    val StringMonoid: Monoid[String] = new {
        def op(o1: String, o2: String): String = o1 + o2
        val id = ""
    }

    val IntAdditionMonoid: Monoid[Int] = new {
        def op(o1: Int, o2: Int): Int = o1 + o2
        val id = 0
    }
```

Notice that we provide the binary operation and the identity element. It is up to us programmers to ensure that the two abide by the monoid laws.

Scala will infer that these monoids are of that type, so there is no need to explicitly declare the type of the new object.

Let's use these to create programs.

By the way: packages I

- Up until now, we have written all our code in one file. For large programs, however, it make sense to split our code into multiple packages. While we are not going into too much detail, we will exemplify the use of packages in this course.

- Our monoid code is declared in the package monoids. We can effectively say that the name of the package is the "name space" of our code. This allows us, amongst other, to distinguish between synonyms.

- Whenever you develop a package, you have to compile that file.

```
$ scalac Monoid.scala
```

- You will notice that folders will be created that contain byte code.

By the way: packages II

- In the REPL environment, we can import these packages as follows
- Import everything from the monoids package (we only have the Monoid trait and companion object)
  - `scala> import monoids._`

- Import the monoid trait and companion object from the monoids package
  - `scala> import monoids.Monoid`

- Import the StringMonoid from the companion object in the
  - `scala> import monoids.Monoid.`
  - `scala> StringMonoid`
  - `val res0: monoids.Monoid[String] = monoids.Monoid$$anon$1@1f48fa72`

- Import two specific objects of the companion object
  - `scala> import monoids.Monoid.{StringMonoid, IntAdditionMonoid}`

- Import everything of the companion object
  - `scala> import monoids.Monoid`
  - `scala> foldMap`
  - `val res1: (List[Any], monoids.Monoid[Any]) => (Any => Any) => Any = ...`

Folding lists with monoids I

- Recall the functions `foldLeft` and `foldRight`:
  - `foldLeft[B](z: B)(op: (B, A) => B): B`
    - Applies a binary operator to a start value and all elements of this sequence, going left to right.
  - `foldRight[B](z: B)(op: (A, B) => B): B`
    - Applies a binary operator to all elements of this list and a start value, going right to left.

- What happens if both type parameters A and B are the same type?
  - `foldLeft(z: A)(op: (A, A) => A): A`
  - `foldRight(z: A)(op: (A, A) => A): A`

- This signature corresponds with our monoid!
- This means we can provide these functions our monoids.

Folding lists with monoids II

```
scala> import monoids.Monoid._
scala> val strings = List("1", "2", "3", "4")
val strings: List[String] = List(1, 2, 3, 4)

scala> val ints = List(1, 2, 3, 4)
val ints: List[Int] = List(1, 2, 3, 4)

scala> strings.foldRight(StringMonoid.id)(StringMonoid.op)
val res7: String = 1234

scala> ints.foldLeft(IntAdditionMonoid.id)(IntAdditionMonoid.op)
val res8: Int = 10
```

Since our monoids abide by the laws of associativity, it does not matter whether we use `foldRight` or `foldLeft`.

Importing packages in *scripts*

Generalizing problems with monoids I

- We have seen that we can use `foldLeft` and `foldRight` with monoids.
- That also means that we can now write functions that generalize problems.

- E.g., we can define a function `combineAll` that folds a list with a monoid.

```
def combineAll[A](l: List[A], m: Monoid[A]) : A =
    l.foldLeft(m.id)(m.op)

----------

scala> combineAll(List(1, 2, 3, 4), IntAdditionMonoid)
val res1: Int = 10

scala> combineAll(List("a", "b", "c"), StringMonoid)
val res2: String = abc
```

Notice how we start solving multiple problems with one solution.

Generalizing problems with monoids II

- What if our list has a type that does not have a monoid instance?
- We can map over the list and turn the elements into a type does.

```
def foldMap[A, B](l: List[A], m: Monoid[B])(f: A => B): B =
    l.foldLeft(m.id)((b, a) => m.op(b, f(a)))
```

If we only provide a list and a monoid, this function returns a lambda that expects a function `A => B` transforming elements of type A in the list into elements of type B corresponding with the monoid.

```
----------

case class Person(name: String, weight: Int)
val people = List(Person("An", 75), Person("Tom", 56), Person("Jo", 83))

----------

scala> foldMap(people, IntAdditionMonoid)(x => x.weight)
val res0: Int = 214
```

Rewriting combineAll

```
def combineAll[A](l: List[A], m: Monoid[A]): A =
    l.foldLeft(m.id)(m.op)

def combineAll2[A](l: List[A])(using m: Monoid[A]): A =
    l.foldLeft(m.id)(m.op)

--------

scala> combineAll2(List(1,2,3,4,5))
15
```

By the way: parallel and distributed computing

- Monoids' operations are associative, which means it does not matter how we fold data structures. Given an associative operation, the following all yield the same result:

  - `op(a,op(b,op(c,d)))` — a right fold
  - `op(op(a,b),op(c,d))` — a balanced fold
  - `op(op(op(a,b),c),d)` — a left fold

- A balanced fold, however, is interesting for parallel and distributed computing.
  - Parallel computing: multiple processes execute tasks simultaneously (not necessarily working on the same problem)
  - Distributed computing: a task is divided across multiple machines
- Notice how a balanced fold allows for both parallel and distributed computing. `op(a,b)` and `op(c,d)` are independent and can thus be computed independently by different process on the same machine or by different machines entirely.
- This gives you an indication why functional programming (data structures, techniques, etc.) is important for data engineering.

Our code so far…

```
package monoids

trait Monoid[A]:
    def op(o1: A, o2: A): A
    val id: A

object Monoid:
    val StringMonoid: Monoid[String] = new:
        def op(o1: String, o2: String): String = o1 + o2
        val id = ""

    val IntAdditionMonoid: Monoid[Int] = new:
        def op(o1: Int, o2: Int): Int = o1 + o2
        val id = 0

    def combineAll[A](l: List[A], m: Monoid[A]): A =
        l.foldLeft(m.id)(m.op)

    def foldMap[A, B](l: List[A], m: Monoid[B])(f: A => B): B =
        l.foldLeft(m.id)((b, a) => m.op(b, f(a)))
end Monoid
```

The dual of a monoid

- Every monoid has a dual. The dual of a monoid is straightforward:

```
def dual[A](m: Monoid[A]): Monoid[A] = new:
    def op(x: A, y: A): A = m.op(y, x)
    val id = m.id
```

- It takes as input a monoid and merely "flips" the arguments of op.
- Remember, monoids have a binary operation that is associative.
  - If the operation is commutative as well, then the dual is equivalent.

- Example of a non-commutative monoid?
  - Function composition, for example. The composition of plus2 and times3 is not the same as the composition of times3 and plus2.

The endo monoid I

- A function having the same argument and return type is called an endofunction for which we can create a monoid.

```
given endoMonoid[A]: Monoid[A => A] with
    def op(f: A => A, g: A => A) = f.andThen(g)
    val id = identity
```

- What does this mean?
  - `op` allows us to combine function of type `A => A`. Functions in Scala have a function `andThen` that allows one to "chain" functions.
  - `id` contains a reference to Scala's `identity` function. This function just returns its input value! :-)

The endo monoid II

```
given endoMonoid[A]: Monoid[A => A] with
    def op(f: A => A, g: A => A) = f.andThen(g)
    val id = identity
```

- How is this useful?
- Well, it allows us to easily chain a bunch of function which we can apply to an argument!

```
scala> import monoids.Monoid.{given, *}

scala> val f = List((x: Int) => x+1, (x: Int) => x*2, (x: Int) => x-5)
val f: List[Int => Int] = List(…, …, …)

scala> val x = combineAll2(f)
val x: Int => Int = scala.Function1$$Lambda$…

scala> x(5)
val res1: Int = 7
```

Part 04
Functions that use monoids

Type classes and context parameters

- Currently, our functions such as `foldMap` take monoids as arguments; we provide specific `Monoid` instances (e.g., `Monoid[Ing]`), which we defined as values, as arguments.

```
def foldMap[A,B](l: List[A], m: Monoid[B])(f: A => B): B =
    l.foldLeft(m.id)((b, a) => m.op(b, f(a)))
```

- Instead of passing monoid values, we can have Scala pass them automatically by altering the definition of `foldMap` using the `using` keyword.

```
def foldMap[A,B](l: List[A])(f: A => B)(using m: Monoid[B]): B =
    l.foldRight(m.id)((a, b) => m.op(f(a), b))
```

- We have moved the monoid parameter to its own parameter list (*) and declared it as a context parameter with the `using` keyword.
  - (*) We can have multiple context parameters in a list.

Type classes and context parameters I

```
def foldMap[A, B](l: List[A])(f: A => B)(using m: Monoid[B]): B =
    l.foldRight(m.id)((a, b) => m.op(f(a), b))
```

- Context parameters are not explicitly passed as an argument. Scala will look for a given instance for each context parameter. Given instances are also known as givens and are defined with the `given` keyword.
- We rewrite out monoids as given instances.

```
given IntAdditionMonoid: Monoid[Int] with
    def op(o1: Int, o2: Int) = o1 + o2
    val id = 0
```

- These give us canonical instances for `Monoid[Int]` and `Monoid[String]`.
- Scala can look find the corresponding givens by looking at the type parameters.

Type classes and context parameters II

- Now, instead of

```
scala> foldMap(people, IntAdditionMonoid)(x => x.weight)
val res0: Int = 214
```

  We can write

```
scala> foldMap(people)(x => x.weight)
val res0: Int = 214
```

- Scala infers that the function passed to `foldMap` returns an `Int` and subsequently searches for a given instance of `Monoid[Int]`.
- By the way: where does Scala look for givens? First in the current scope (locally defined or imported instances) and then in the companion objects (both of our `Monoid` or that of `Int`).

Anonymous givens

- When you are defining givens are they are not referred to in your code by name, then you can leave out the name. These anonymous values will be given an "internal name" by Scala.
- You can write

```
given StringMonoid: Monoid[String] with
    def op(o1: String, o2: String) = o1 + o2
    val id = ""
```

  as

```
given Monoid[String] with
    def op(o1: String, o2: String) = o1 + o2
    val id = ""
```

- Anonymous objects are outside the scope of the course.

Type classes and context parameters III

- **We cannot have two givens for** `Monoid[Int]`. If other `Monoid[Int]` objects need to be defined, then they need to be explicitly passed as parameters. For example, I add the following `Monoid[Int]` next to my `IntAdditionMonoid`.

```
val IntProductMonoid: Monoid[Int] = new:
    def op(o1: Int, o2: Int): Int = o1 * o2
    val id = 1
----------
scala> import monoids.Monoid.{given, *}
scala> foldMap(people)(x => x.weight)
ambiguous given instances: both object IntProductMonoid in
object Monoid and object IntAdditionMonoid in object Monoid
match type monoids.Monoid[Int] of parameter m of method foldMap
in object Monoid

scala> foldMap(people)(x => x.weight)(using IntProductMonoid)
val res1: Int = 348600
```

Type classes: a summary

"A type class is an abstract, parameterized type that lets you add new behaviour to any closed data type without using sub-typing." [1] This is usually done with Traits in Scala. For example: we have seen how we have created a function `sum` for `List[Int]`.

Here, we have created a trait `Monoid[A]` and canonical instances of that trait for different types (`IntAdditionMonoid` for `Int`, `StringMonoid` for `String`, …). The use of givens allows us to avoid passing `Monoid`s as values, as Scala's type system can figure out which one to use thanks to using clauses [2].

1. https://docs.scala-lang.org/scala3/book/types-type-classes.html
2. https://docs.scala-lang.org/scala3/book/ca-given-using-clauses.html

Part 05
Foldable

Preamble: Foldable

- Foldable is a class of ADTs that can be folded to a value.
  - We can use arbitrary functions to fold elements into one value.
  - This is thus more general than a Monoid!

- Foldable objects must implement ~~`fold or (inclusive or)`~~ `foldMap` that operate on Monoids.
  - `def foldMap[B](f: A => B)(using m: Monoid[B]): B`
  - *Scala already defines fold for many ADTs. In FP, "`fold`" is usually reserved for Foldable objects and operates on Monoids. We will focus on `foldMap` since only one of the two needs to be provided.*

- They can also provide functions that solve more general problems:
  - `def foldRight[B](z: B)(f: (A, B) => B): B`
  - `def foldLeft[B](z: B)(f: (A, B) => B): B`
  - `def fold[B](z: B)(f: (A, B) => B): B`

- Important is that all these functions can be implemented with `foldMap`!

Foldable data structures I

- In this course, we have implemented a couple of data structures that can be folded: `List`, `Tree`, `Flux`, and `Option`. When processing data contained in these data structures, should we care about the the data structure? No.
- If `ints` is a data structure containing `Int`, we would like to obtain the sum using `ints.foldRight(0)(_ + _)` no matter the data structure.

```
scala> Branch(Branch(Leaf(1),Leaf(2)), Leaf(3)).foldRight(0)(_ + _)
val res0: Int = 6

scala> Some(1).foldRight(0)(_ + _)
val res1: Int = 1
```

- In Scala, we can capture this commonality in a trait.

Foldable data structures II

- In the next few slides, we will capture these commonalities for `Tree` and our implementation of `Option`.

`Tree` does not exist in Scala's standard library, but `Option` does. Moreover, `Option` in Scala's standard library has already implemented map, foldRight, foldLeft, etc. We are extending our `Option` to exemplify FP.

- We assume the availability of the following packages:

```
package trees

sealed trait Tree[+A]
case class Leaf[A](value: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
----------
package option

sealed trait Option[+A]
case class Some[+A](get: A) extends Option[A]
case object None extends Option[Nothing]
```

Trait Foldable I

```
package foldable

import monoids.{given, *}

trait Foldable[F[_]]:
    import Monoid.{endoMonoid, dual}

    extension [A](as: F[A])
        // ...
```

- Much like we can abstract over types, we can abstract over type constructors. We call these higher-kinded types.
- `F[_]` matches any type constructor that takes one argument such as `List[Int]` and `Tree[String]`.

- This informs the language that we are going to extend objects that match that type constructor.
- We are going to define function such as foldRight, foldLeft, etc.

Trait Foldable II

```
trait Foldable[F[_]]:
    import Monoid.{endoMonoid, dual}

    extension [A](as: F[A])
        def foldRight[B](acc: B)(f: (A, B) => B): B =
            as.foldMap(f.curried)(using endoMonoid[B])(acc)

        def foldLeft[B](acc: B)(f: (B, A) => B): B =
            as.foldMap(a => b => f(b, a))(using dual(endoMonoid[B]))(acc)

        def foldMap[B](f: A => B)(using mb: Monoid[B]): B =
            as.foldRight(mb.id)((a, b) => mb.op(f(a), b))

        def combineAll(using ma: Monoid[A]): A =
            as.foldLeft(ma.id)(ma.op)

        def toList: List[A] =
            as.foldRight(List.empty[A])(_ :: _)
```

Notice how we have already implemented all these functions, but… there's a circular dependency?

`foldRight` depends on `foldMap`.
`foldMap` depends on `foldRight`.

Indeed, our givens will need to implement either `foldMap` or `foldMap` that does not rely on the other.

Trait Foldable III

```
def foldMap[B](f: A => B)(using mb: Monoid[B]): B =
    as.foldRight(mb.id)((a, b) => mb.op(f(a), b))
```

Assuming `foldRight` is available as a member of our object, the definition of `foldMap` is straightforward; Scala will look for the Monoid of type `B`, and then use that monoids `id` and `id`.

```
def combineAll(using ma: Monoid[A]): A =
    as.foldLeft(ma.empty)(ma.combine)
```

`combineAll` is written in function of `foldMap`. Our object must have either `foldMap` or `foldRight` for it to work. We could have also written this function using `foldRigt`.

Trait Foldable IV

```
def foldRight[B](acc: B)(f: (A, B) => B): B =
    as.foldMap(f.curried)(using endoMonoid[B])(acc)
```

Assuming `foldMap` is available as a member of our object.

- `foldRight` takes as input an `(A, B) => B`.
- But `foldMap` requires a `A => B`.
- We curry to turn the `(A, B) => B` into a `A => (B => B)`.
- We "transform" our `A => (B => B)` objects into `(B => B)` objects.
- This means we can can use the `endoMonoid[B]` to compose them.
- The result of the `foldMap` is a function that we apply to `acc`.

Trait Foldable V

```
def foldLeft[B](acc: B)(f: (B, A) => B): B =
    as.foldMap(a => b => f(b, a))(using dual(endoMonoid[B]))(acc)
```

Assuming `foldMap` is available as a member of our object.

1. `foldLeft` takes as input an `(B, A) => B`, but `foldMap` requires a `A => B`
2. The `foldMap` takes a function ensuring arguments in the right order.
3. And the dual of endoMonoid ensures that the functions are also composed in the right way.

Implementing the trait Foldable I

```
object Foldable:
    import trees.{Tree, Leaf, Branch}
    given Foldable[Tree] with
        extension [A](ds: Tree[A])
            def foldMap[B](f: A => B)(using m: Monoid[B]) = ds match
                case Leaf(a) => f(a)
                case Branch(l, r) => m.op(l.foldMap(f), r.foldMap(f))

    import option.{Option, Some, None}
    given Foldable[Option] with
        extension [A](ds: Option[A])
            def foldMap[B](f: A => B)(using m: Monoid[B]) = ds match
                case None => m.id
                case Some(a) => f(a)
end Foldable
```

Implementing the trait Foldable II

```
object Foldable:
    import trees.{Tree, Leaf, Branch}
    given Foldable[Tree] with
        extension [A](ds: Tree[A])
            def foldMap[B](f: A => B)(using m: Monoid[B]) = ds match
                case Leaf(a) => f(a)
                case Branch(l, r) => m.op(l.foldMap(f), r.foldMap(f))

    import option.{Option, Some, None}
    given Foldable[Option] with
        extension [A](ds: Option[A])
            def foldMap[B](f: A => B)(
                case None => m.id
                case Some(a) => f(a)
end Foldable
```

- We define givens for context parameters.
- Here, we provide an instance for Foldable[Tree].
- This given instance will provide an implementation for Tree.

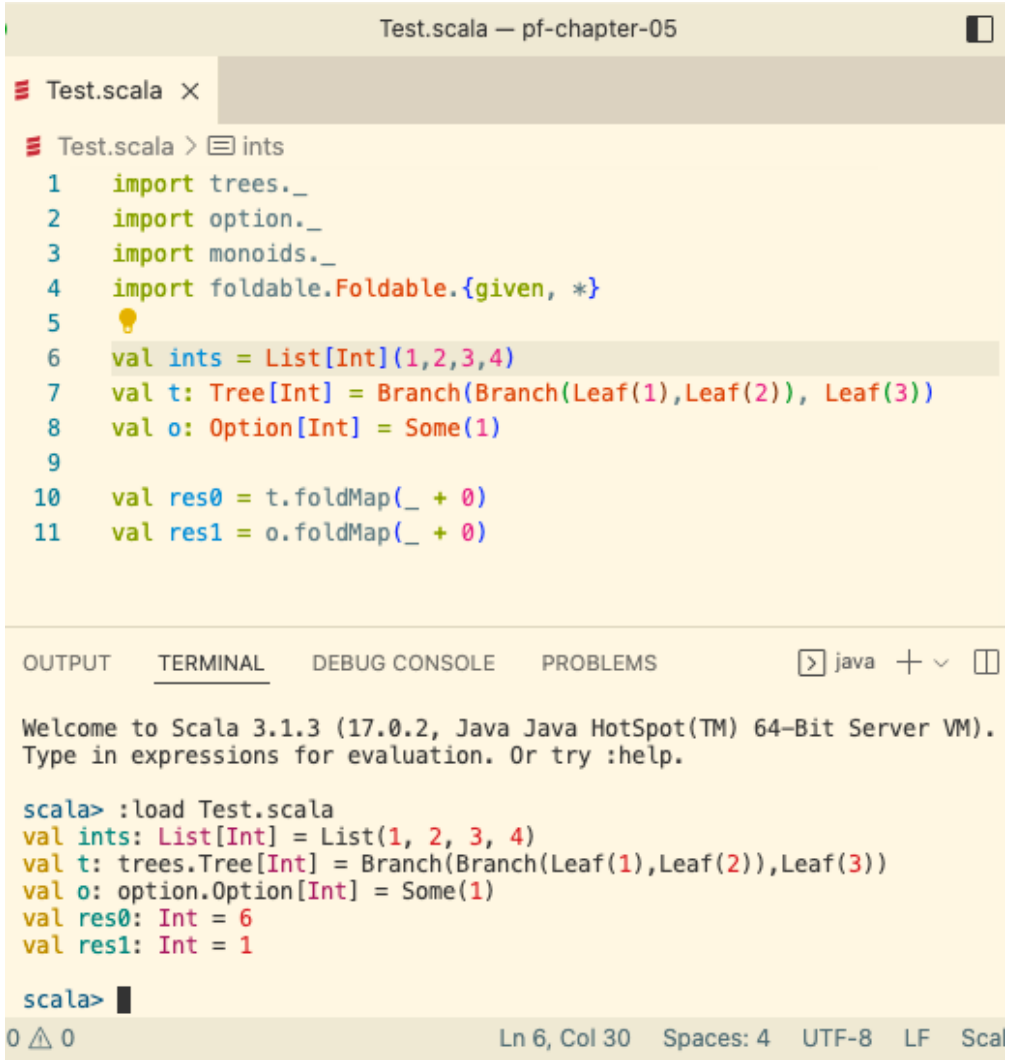
Implementing the trait Foldable III

```
object Foldable:
    import trees.{Tree, Leaf, Branch}
    given Foldable[Tree] with
        extension [A](ds: Tree[A])
            def foldMap[B](f: A => B)(using m: Monoid[B]) = ds match
                case Leaf(a) => f(a)
                case Branch(l, r) => m.op(l.foldMap(f), r.foldMap(f))
```

- This is a technicality, but we also need to specify that we are going to extend objects of type Tree[A]. Scala knows that this is a Foldable object, and therefore will know that we are implementing the function foldMap for Tree.
- By the way: Scala "knows" we are overriding this function's signature. We can add the **override** keyword, but that's not necessary. You will learn more about this in OOP.

Using our Foldable trait

- Notice that I had to import all the packages that I needed.
- Importing _ or * imports all named objects, but not the givens. To import givens, one must use the keyword `given`.
- You can now see how I can apply the foldMap function to both Tree and Option without worrying how they are implemented.



By the way

- In the book, the authors propose that the trait provides implementations for all functions. If a specific implementation does not implement one of these, this will lead to an infinite recursion. They furthermore use some advanced tricks through advanced exercises.

```
trait Foldable[F[_]]:
  import Monoid.{endoMonoid, dual}
  extension [A](as: F[A])
    def foldRight[B](z: B)(f: (A, B) => B): B =
        as.foldMap(f.curried)(using endoMonoid[B])(z)

    def foldLeft[B](z: B)(f: (B, A) => B): B =
        as.foldMap(a => b => f(b, a))(using dual(endoMonoid[B]))(z)

    def foldMap[B](f: A => B)(using m: Monoid[B]): B =
        as.foldRight(m.id)((a, b) => m.op(f(a), b))
```

foldRight depends on foldMap and vice verse. If one of these functions is not overwritten in an appropriate way, we would have an infinite recursion.

- The idea is that it is up to *the programmer to choose which functions to override* (e.g., because a programmer knows which one is more efficient). The other functions are then supported via the overwritten ones.
- **This is outside the scope of this course, but I am happy to discuss their code outside class.**

Part 06
Composing monoids

Composing monoids

- We will complete the other functions in the exercises.

- So far, we have seen the properties of monoids and how they can be used to generalize foldMap.

- This may seem not that interesting, but monoids become interesting when they are composed and assembled to create complex monoids. We will cover these two in the next slides.

Product monoids I

- If A and B are monoids, then the tuple type (A, B) is also a monoid. We call this the product of A and B.

```
given productMonoid[A, B](
        using m1: Monoid[A], m2: Monoid[B]
    ): Monoid[(A, B)] with
    def op(o1: (A, B), o2: (A, B)): (A, B) =
        (m1.op(o1(0), o2(0)), m2.op(o1(1), o2(1)))
    val id = (m1.id, m2.id)
```

For the 1st element, we apply the first operation on the As. For the 2nd element, we apply the second operation on the Bs.

Product monoids II

- We can now perform multiple calculations using one pass!

```
scala> import monoids.Monoid.{given, *}
scala> val t = Branch(Branch(Leaf(1),Leaf(2)), Leaf(3))
val t: trees.Tree[Int] = Branch(Branch(Leaf(1),Leaf(2)),Leaf(3))

scala> val data = t.foldMap((1, _))
val data: (Int, Int) = (3,6)

scala> val mean = data(1) / data(0).toDouble
val mean: Double = 2.0
```

Monoids on data structures

- So far, we have seen monoids for value types.
- Are monoids restricted to value types? No!
- For instance, some data structures form monoids if the type of its elements also form monoids.

- Let's define a monoid for merging associative arrays, also known as maps.

mapMergeMonoid I

```
given mapMergeMonoid[K, V] (using m: Monoid[V]):
    Monoid[Map[K, V]] with
        def op(a: Map[K, V], b: Map[K, V]) =
            (a.keySet ++ b.keySet).foldLeft(id) {
                (z, k) =>
                  z.updated(
                    k,
                    m.op(a.getOrElse(k, m.id), b.getOrElse(k, m.id))
                  )
            }
        val id = Map()
```

- We combine the two key sets into a new key set and then fold it with foldLeft.
- We will construct a new map, so our accumulator is the empty map (see highlight).
- If values for a key exist in either map, we use these values. Otherwise, the id of the monoid is used as a default (and neutral) value.

mapMergeMonoid II

- Let's use mapMergeMonoid. We will now use summon to retrieve a given that corresponds with the type specified. summon is a function that is available in Scala and requires the use of square brackets.

```
val a = Map("i1" -> 1, "i2" -> 2)
val b = Map("i2" -> 3, "i3" -> 5)

----------

scala> val m = summon[Monoid[Map[String, Int]]]
val m: monoids.Monoid.mapMergeMonoid[String, Int] = ...

scala> m.op(a,b)
val res2: Map[String, Int] = Map(i1 -> 1, i2 -> 5, i3 -> 5)
```

Notice how Scala can determine which monoids to use by analyzing types.

Whenever a monoid is unavailable or there is an ambiguity, Scala will throw an exception.

Summary

- We have looked at algebraic data types and our first purely algebraic structure: monoids

- We have used monoids to implement our own library for foldable data structures. Foldable data structure are import in functional programming.

- A fold allows us to process data in data structures without worrying about the specific data structure. This is possible thanks to monoids, which abide by monoid laws: associativeness and identity.

Lexicon

- Associative array – tableau associatif
  - Also known as a *map*
- Combinator – combinateur
- Context parameter – paramètre de contexte
- Distributed computing – calcul distribué
- Given instance – instance donnée
- Parallel computing – parallélisme