INFO0054 Programmation Fonctionnelle – Exercises

Christophe Debruyne

Exercises 8: Monoids and Foldable

In these exercises, we will use the following ADTs. In these exercises, we will declare everything in one file. You are invited to separate the various parts into different files and packages. You can find code pertaining to Monoid and Foldable in the Appendix.

```scala
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]

sealed trait Option[+A]:
    def map[B](f: A => B): Option[B] = this match
        case None => None
        case Some(a) => Some(f(a))

    def getOrElse[B>:A](default: => B): B = this match
        case None => default
        case Some(a) => a

    def orElse[B>:A](ob: => Option[B]): Option[B] =
        map(Some(_)).getOrElse(ob)

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

Exercise 1:

Given our trait Monoid, create monoids for Boolean Or, Boolean And, and List concatenation. Demonstrate their use in the REPL environment.

Solution 1:

```scala
    given ListMonoid[A]: Monoid[List[A]] with
        def op(o1: List[A], o2: List[A]) = o1 ++ o2
        val id = Nil

    given BoolAndMonoid: Monoid[Boolean] with
        def op(o1: Boolean, o2: Boolean) = o1 && o2
        val id = true

    given BoolOrMonoid: Monoid[Boolean] with
        def op(o1: Boolean, o2: Boolean) = o1 || o2
        val id = true
```

When Scala must choose between monoids, Scala will issue an error about ambiguous instances. In that case, it is up to us to explicitly state which monoid to use. Since monoids are declared as context parameters, we must pass those explicit references with the `using` keyword.

```
scala> combineAll(List(List(1,2,3),List(4,5),List(),List(6)))
val res3: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> foldMap(List(1,2,3,4))(_ >=2)
-- Error: ----------------------------------------------------------------------
1 |foldMap(List(1,2,3,4))(_ >=2)
  |                               ^
  |Ambiguous given instances: both object BoolOrMonoid in object Monoid and
   object BoolAndMonoid in object Monoid match type Monoid[Boolean] of parameter m
   of method foldMap in object Monoid
1 error found

scala> foldMap(List(1,2,3,4))(_ >=2)(using BoolOrMonoid)
val res4: Boolean = true

scala> foldMap(List(1,2,3,4))(_ >=2)(using BoolAndMonoid)
val res5: Boolean = false
```

Exercise 2:

Given our trait Monoid, create monoids for combining Option objects. What is the identity element? What is the binary operation? Are there different ways to combine Option objects? What can you tell about the binary operation?

Solution 2:

The identity element for combining option elements is None. There are two ways to combine Option objects with `orElse`: `x.orElse(y)` or `y.orElse(x)`. The function `orElse` is associative and together with the identity element satisfies the monoid laws. The function `orElse` is not associative, meaning that `x.orElse(y)` or `y.orElse(x)` yield different results. We are, however, free to choose.

Version 1:

```
    given OptionMonoid[A]: Monoid[Option[A]] with
        def op(oa: Option[A], ob: Option[A]) = oa.orElse(ob)
        val id = None
```

```
scala> combineAll(List(None,Some(1),Some("foo"),None))
val res0: Option[Int | String] = Some(1)
```

Version 2:

```
    given OptionMonoid[A]: Monoid[Option[A]] with
        def op(oa: Option[A], ob: Option[A]) = ob.orElse(oa)
        val id = None
```

```
scala> combineAll(List(None,Some(1),Some("foo"),None))
val res0: Option[Int | String] = Some(foo)
```

Exercise 3:

Monoids can be composed. This means that if types A and B are monoids, then the tuple type (A, B) is also a monoid. We call this the product of monoids. We have defined such a monoid in class.

```scala
    given productMonoid[A, B](using ma: Monoid[A], mb: Monoid[B]): Monoid[(A, B)] with
        def op(x: (A, B), y: (A, B)) = (ma.op(x(0), y(0)), mb.op(x(1), y(1)))
        val id = (ma.id, mb.id)
```

Use this monoid to define a new function `combineAll` that takes as input a `Map[A,B]`. This function, which bears the same name as `combineAll` operating on lists, "fetches" the product monoid of A and B. This exercise also demonstrates that you can have functions with the same name as long as there's a difference in their inputs.

```scala
scala> val x = Map("x" -> 24, "y" -> 25, "z" -> 26)
val x: Map[String, Int] = Map(x -> 24, y -> 25, z -> 26)

scala> val y = combineAll(x)
val y: (String, Int) = (xyz,75)

scala> val a = Map("x" -> List(24), "y" -> List(25), "z" -> List(26))
val a: Map[String, List[Int]] = Map(x -> List(24), y -> List(25), z -> List(26))

scala> val b = combineAll(a)
val b: (String, List[Int]) = (xyz,List(24, 25, 26))
```

Solution 3:

```scala
    def combineAll[A,B](as: Map[A,B])(using m: Monoid[(A, B)]): (A,B) =
        as.foldLeft(m.id)(m.op)
```

Exercise 4:

List is a foldable data structure and already has implemented many of the methods of our Foldable. List does not have a method `combineAll`, however. Declare List as a foldable so that objects of that type have access to such a method. Lists have a function `toList`, but make sure that we avoid any unnecessary computations when using the given to compute `toList`.

Solution 4:

```scala
    given Foldable[List] with
        extension [A](as: List[A])
            override def foldRight[B](acc: B)(f: (A, B) => B) =
                as.foldRight(acc)(f)
            override def toList: List[A] = as
```

```scala
scala> List(1,2,3,4).combineAll
val res0: Int = 10

scala> List(1,2,3,4).toList
val res1: List[Int] = List(1, 2, 3, 4)

scala> val f = summon[Foldable[List]]
val f: Foldable.given_Foldable_List.type = Foldable$given_Foldable_List$@71e88441

scala> f.toList(List(1,2,3,4))
toList of Foldable[List]
val res2: List[Int] = List(1, 2, 3, 4)
```

Exercise 5:

Create a given that provides objects of LBranch the Foldable trait.

```
sealed trait LTree[+A]
case class LLeaf[A](value: A) extends LTree[A]
case class LBranch[A](left: LTree[A], value: A, right: LTree[A]) extends LTree[A]
```

```
scala> val t = LBranch(LBranch(LLeaf("a"), "b", LLeaf("c")), "d", LLeaf("e"))
val t: LBranch[String] = LBranch(LBranch(LLeaf(a),b,LLeaf(c)),d,LLeaf(e))

scala> t.combineAll
val res0: String = abcde

scala> t.toList
val res1: List[String] = List(a, b, c, d, e)
```

Solution 5:

```
    given Foldable[LTree] with
        extension [A](ds: LTree[A])
            override def foldMap[B](f: A => B)(using m: Monoid[B]) = ds match
                case LLeaf(a) => f(a)
                case LBranch(l, v, r) => m.op(l.foldMap(f), m.op(f(v), r.foldMap(f)))
```

References

[1] Paul Chiusano and Rnar Bjarnason. 2015. Functional Programming in Scala (2nd. ed.). Manning Publications Co., USA.

A Monoid

```scala
trait Monoid[A] {
    def op(o1: A, o2: A): A
    val id: A
}

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

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

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

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

    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))

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

B Foldable

```scala
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])(_ :: _)

object Foldable:
    given Foldable[Tree] with
        extension [A](ds: Tree[A])
            override 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))

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