Chapter 40
Scalaz

40.1 Introduction

If you start working with Scala you will probably quickly come across the library Scalaz. It is a very widely used set of extensions to the core language. In many cases when people talk about developing systems in Scala they actually mean a combination of Scala and Scalaz.

Scalaz is actually a very large library of extensions which could have (and do have) books dedicated just to it. Indeed, if you look at

- http://eed3si9n.com/learning-scalaz

You can find a 21-day course dedicated to Scalaz. As this book is intended as an introduction to Scala and covers much of the language, this chapter will be restricted to an introduction to Scalaz and an overview of the most useful features for you at this stage.

40.2 Obtaining Scalaz

Scalaz is not a standard part of the Scala installation. You must therefore add it yourself to your environment. At the time of writing, the current version of Scalaz is 7.2.17 for Scala 2.12.

The easiest way is to add it to your project dependencies using Maven or SBT. To add Scalaz to your Maven project use:

```
<dependency>
    <groupId>org.scalaz</groupId>
    <artifactId>scalaz-core_2.12</artifactId>
    <version>7.2.17</version>
</dependency>
```


J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_40

If you are using SBT, add the following line to your build file:

```
libraryDependencies += "org.scalaz" %% "scalaz-core" % "7.2.17"
```

Alternatively, you can download Scalaz from

- http://central.maven.org/maven2/org/scalaz/scalaz-core_2.12/7.2.17/scalaz-core_2.12-7.2.17.jar

40.3 Scalaz Overview

Scalaz is a very large and exhaustive library of extensions for Scala. Its focus is on emphasizing and supporting functional programming and type correctness. Many teams therefore adopt Scalaz either because they wish to explicit its set of typesafe extensions or because they wish to further push the use of functional program (or of course both).

A significant aspect of Scala is its support for typesafe operations. These are primarily supported by the introduction of Typeclasses in Scalaz. A Typeclass is a mechanism for ad hoc, compile time, polymorphism. Note the comment about 'compile time' in that sentence. In standard Scala (and languages such as Java and C#) runtime polymorphism is supported. However, Scalaz supports compile time polymorphism, while the distinction may not be of that much interest to you; it does mean that at compile time you can be notified of potential issues that might only be identified at runtime due to undesirable behaviour. An example might be:

```
1 == "1"
```

This is legal Scala but will always return false; as an Int can never equal a String. In Scalaz the '===' operator is used to perform a compile time check on the types. Thus

```
1 == "1"
```

will generate a compile time error (rather than allowing the runtime environment to merely execute).

From a functional programming point of view, the extensions provided by Scalaz are inspired by Haskell and the theoretical concepts behind functional programming. This can mean that, at least initially, it can seem that Scalaz is complicated and quiet abstract (which is probably true) although some aspects of the library are very straight forward to use and very useful (and these are the focus of this chapter).

There are many tutorials available online which will explore much of the theoretical side of Scala such as Monads, Functors, Applicatives, etc., and some of these are listed at the end of this chapter.

40.4 Some Useful Typeclasses

40.4.1 Equals Typeclass

Scalaz provides a typesafe equals operator that will generate a compile error if incompatible types are compared.

For example, while `==` and `!=` in standard Scala will allow you to compare a String and an Int but will always return false, the Scalaz operators `===` and `=/=` operators will result in a compile time error:

```
object EqualsApp extends App {

  println(s"1 == 1: ${1 == 1}")
  println(s"1 === 1: ${1 === 1}")
  // Always returns false - compiler gives warning
  println(s"1 == '1': ${1 == "1"}")
  // won't even compile - Type mistach
  // println(1 === "1")
}
```

As can be seen above although Scala allows the integer 1 and the string “1” to be compared, Scalaz does not.

40.4.2 Order Typeclass

Scalaz provides a very easy to use typeclass that provides for typesafe ordering. This means that instances can be compared for relative ordering but that ordering of incompatible types will now generate a runtime error:

```
object OrderApp extends App {
  // Standard Scala ordering operators
  println(s"1 < 4d: ${1 < 4d}")
  // println(1 lte 4d) // won't compile type mismatch
  println(s"1 ?|? 1: ${1 ?|? 1}")
  println(s"1 ?|? 2: ${1 ?|? 2}")
  println(s"2 ?|? 1: ${2 ?|? 1}")
  (1 ?|? 2) match {
    case Ordering.LT => println("Its less than")
    case Ordering.EQ => println("They are Equal")
    case Ordering.GT => println("Its greater than")
  }
  // println(1 ?|? 2d) // won't compile type mismatch
}
```

In the above example Scala allows the integer 1 and the double 4 to be compared, whereas Scalaz does not.

The `?|?` operator can also be used to compare two compatible values and returns a `scalaz.Ordering` object. This object is one of:

- `Ordering.LT`—indicating that the first value is less than the second value
- `Ordering.EQ`—indicating that the values are equal
- `Ordering.GT`—indicating that the second value is greater than the first.

The output generated from this application is:

```
1 < 4d: true
1 ?|? 1: EQ
1 ?|? 2: LT
2 ?|? 1: GT
Its less than
```

40.5 Standard Class Extensions

40.5.1 Extensions to Option

Scalaz provides a set of extensions to the `Option` type in Scala. These extensions make it easier to work with Options. For example, there are some convenience constructor methods that return `Option` types (rather than returning Some or None types). See

- `some`—returns a Some instance but the specified type is `Option[T]`
- `none[T]`—returns a None value of type `Option[T]`

Scalaz also provides some alternatives to the `getOrElse` operation available on standard Options types. These such as the ternary operator for Options do not directly have an equivalent in the standard library.

```
object OptionApp extends App {
  // Support to make working with Options easier

  // Traditional Scala – return specific types
  // Can cause problems with code expecting an Option[T]
  val o1: Some[Int] = Some(10)
  val o2: Option[Nothing] = None

  // Scalaz Scala – both operations return Option[T]
  val s1: Option[Int] = some(10)
  val s2: Option[Int] = none[Int]

  // Alternative for getOrElse
  val x1 = some(20) | 10
  println(s"x1: $x1")
  val x2 = none | 10
  println(s"x2: $x2")

  // Ternary operator for options
  val x3 = Some(10) ? 5 | 4
  println(s"x3: $x3")

  // ~ operator: Returns the item contained in the Option
  // or its zeroth value
  val someInt = some[Int](3)
  println(~someInt)
  val emptyOption = none[Int]
  println(~emptyOption)

}
```

The final operator examined here is the '∼' operator that returns the item contained in the `Option` (if it is defined), otherwise it returns the Zeroth value for the type specified. If the type is an `Int`, then it is `0`, if it is a `Boolean`, then it is false, etc.

The output from this simple application is:

```
x1: 20
x2: 10
x3: 5
3
0
```

40.5.2 Boolean Extensions

Scalaz also adds some additional functionality to the `Boolean` type. For example, it provides the `?|` and `??` operators.

```
object BooleanApp extends App {

  // Ternery operator for Scala
  println(true ? "This is true" | "This is false")
  println(false ? "This is true" | "This is false")

  // Returns the given argument if this is `true`,
  // otherwise, the zero element for the type of the given
  // argument
  println(false ?? List(1, 2, 3))
  println(true ?? List(1, 2, 3))

}
```

The `?|` operator is a ternary operator that will return the first value following the '?' if the condition is true, otherwise it returns the second value. For example,

- `true ? "This is true" | "This is false")` – returns `"This is true"`
- `false ? "This is true" | "This is false")` – returns `"This is false")`

The ?? operator returns the given argument if the condition is true, otherwise it returns the Zeroth version of the argument, thus

- `true ? "This is true" | "This is false")` – returns `"This is true"`
- `false ? "This is true" | "This is false")` – returns `"This is false")`

40.5.3 Extensions to List

Scalaz also provides some extensions to the List type. Some of these extensions are shown below.

```
object ListExampleApp extends App {

  // get the tail as an option
  val list1 = List(10, 20, 30)
  println(list1.tailOption)

  //intersperse the list with additional elements
  val list2 = list1.intersperse(1)
  println(list2)

  // from list to List[List] of all possibilities
  val list3 = List('a', 'b', 'c').powerset
  println(list3)

}
```

In the above example three extensions to the List type are shown, namely:

- tailOption—this returns a list containing the tail of the list it is applied to wrapped in an Option.
- Intersperse—this adds the provided argument to the elements in the original list, between each of the values held in the list.
- Powerset—this returns a list of lists that represent all combinations of the values in the original list.

Note that none of these operations affects the original list; they all generate a new list.

The output from the example is:

```
Some(List(20, 30))
List (10, 1, 20, 1, 30)
List (List(a, b, c), List(a, b), List(a, c), List(a), List(b, c), List
(b), List(c), List())
```

40.5.4 Extensions for Map

Another collection type that is extended by Scalaz is `Map`. As an example, the following code example illustrates altering a map within a safe manner. The example creates a simple Map and then uses the Scalaz alter operation to modify the value with the key "a". In this case a function is applied to the value in which the value 5 is applied to the existing value. It uses the `|+|` which is an alias for `mappend` (a function that appends a value and returns the result). The `mappand` function takes two values of the same type and returns a value of that type (which is the result of appending them). In this case the argument `f` to the function is of type `Option[Int]`.

```
object MapExampleApp extends App {

  // Alter one entry in the map
  val m1 = Map("a" -> 10, "b" -> 20)
  println(s"m1: $m1")
  val m2 = m1.alter("a")(f => f |+| some(5))
  println(s"m2: $m2")

  // Find the interaction between two maps
  val m3 = Map("b" -> 250, "c" -> 300)
  println(s"m3: $m3")
  val m4 = m1.intersectWith(m3)((m1v, m3v) => m3v)
  println(s"m1.intersectWith(m3)((m1v,m3v) => m3v): $m4")
}
```

The second example in the above code uses a method `intersectWith` that determines the intersection (between keys) of two maps. In this case only the key "b" is common to the maps m1 and m3. The result returned in this case is the value held in the second map. If we wanted the value held in the first map, then we would return that value from the function passed to the `intersectWith` method, for example,

```
val m4 = m1.intersectWith(m3)((m1v, m3v) => m1v)
```

The output from this example is:

```
m1: Map(a -> 10, b -> 20)
m2: Map(a -> 15, b -> 20)
m3: Map(b -> 250, c -> 300)
m1.intersectWith(m3)((m1v,m3v) => m3v): Map(b -> 250)
```

40.5.5 Extensions to String

Scalaz also provides some interesting extensions to the String class.

One example is its introduction of the plural operator which tries (with some mixed success) to convert words into their plural version, for example “Day” into “Days”. It is a little naïve as the approach taken is to pluralise a String by appending an "s" unless this String ends with "y" and not one of ["ay", "ey", "iy", "oy", "uy"] in which case the 'y' character is chopped and "ies" is appended. It is possible to specify an integer to plural that indicates how many of something is being specified; 1 indicates a singular value (and thus just returns the string as is).

```
object StringExampleApp extends App {
  val s1 = "Day"
  println(s"s1.plural(2): ${s1.plural(2)}")
  // Safely parse booleans, bytes, shorts, longs,
  // floats, double and ints
  println("10".parseDouble)
  println("john".parseDouble)
}
```

A more useful feature of the string extensions is the set of parse operations. These will safely parse booleans, bytes, shorts, longs, floats, doubles and ints from a string without throwing an exception if the value passed to them is not a number. Instead the operations return a Success or Failure object that wraps the result or the associated exception.

The output generated by the example application is:

```
s1.plural(2): Days
Success(10.0)
Failure(java.lang.NumberFormatException: For input string: "ten")
```

40.6 The Other Either

Another feature of Scalaz is its provision of what is sometimes known as the *other Either*. Scala provides Either as a way of capture that either a successful result has been generated or not. Scalaz version provides a set of utility methods that make it easier to work with.

The Scalaz Disjunction is defined as `\/[A, B]`. It is right biased (as right by convention is the success element) and provides operations such as `map`, `flatMap`, etc., that work on the right side of the object.

```
object OtherEitherApp extends App {
  // An alternative to Scala's Either
  val r1 = \/.fromTryCatchNonFatal{ "1234".toInt }
  println(r1)
  val r2 = \/.fromTryCatchNonFatal{ "John".toInt }
  println(r2)
}
```

The output from this simple example is:

```
\/-(1234)
-\/(java.lang.NumberFormatException: For input string: "John")
```

One of the easiest ways of working with Sclacz `\/[A, B]` is to use the `fromTryCatchNonFatal` method. This will try to execute an expression. If it is successful, then the result is stored in the *right* side of the result. If an exception is thrown, then this is stored in the *left* side. The side of the value is indicated by a '–'; either to the right or left of the `\/` symbol (as shown above).

40.7 Tagging

Tags can be used to create new types based on existing types. It uses the `@@` symbol to *tag* an existing type as another type (creating a new type). This can be useful if you want to use, for example, a `String` to represent an ID but do not want just any old string to be used.

The following example illustrates the basic idea of using tagging to create a new type:

```
object TagExampleApp extends App {
  class MyId

  type Id = String @@ MyId
  def Id(s: String) = s.asInstanceOf[Id]

  // Does not compile as typs are not the same
  // val y: Id = "M123"

  // Does compile as x is of type Id
  val x: Id = Id("M123")
  println(x)

}
```

In this example, we use the `@@` symbol to tag a String with the class `MyID`. This is used to define a type `Id` (which is actually an alias to `String @@ MyId`, which expands to `@@[String, MyId]`, which in turn expands to `String with Tagged[MyId]`).

Online Scalaz Resources

http://central.maven.org/maven2/org/scalaz/scalaz-core_2.12/7.2.17/scalaz-core_2.12-7.2.17.jar download Scalz JAR file
http://scalaz.github.io/scalaz/#scaladoc ScalaDoc for Scalaz
http://eed3si9n.com/learning-scalaz 21-day online course for Scalaz
http://typelevel.org/blog/2013/10/13/towards-scalaz-1.html Towards Scalaz Tutorial