Algebraic Data Types and Structural Recursion in Scala 2

Algebraic data types are the main way of defining data in Scala. Structural recursion is a general pattern for transforming an algebraic data type.

Here we'll look at both.

Algebraic Data Types

Apple designs computers in California. A MacBook Pro, their popular line of laptops, can be a MacBook Pro 13", MacBook Pro 14", or a MacBook Pro 16". Different models have different options for RAM size and SSD capacity. For example, the 13" model is available with 8GB or 16GB of RAM, and a 256GB, 512GB, 1TB, or 2TB SSD.

Think about how we might represent a MacBook Pro as a data structure in software. The connections between the elements that describe a Macbook Pro are logical ors and logical ands. A Macbook Pro is 13" **or** 14" **or** 16". A 13" model has a specific choice of RAM **and** a specific choice of SSD. This tells us that we can represent a Macbook pro as an algebraic data type.

An *algebraic data type* is any description of data that connections the individual elements with *logical ands* and *logical ors*. If it uses just logical ands we can call it a *product type*. If just logical ors are used, it can be called a *sum type*.

Once we recognize that data is an algebraic data type we can immediately translate it to Scala. There are two patterns, one for logical ors and one for logical ands.

For logical ors:

```scala
// A is a B or C
sealed trait A
final case class B() extends A
final case class C() extends A
```

For logical and:

```scala
// A is a B and C
final case class A(b: B, c: C)
```

Lets return to our Macbook Pro example. A Macbook Pro is 13" **or** 14" **or** 16". We can represent this by taking the pattern for logical ors, and replacing `A`, `B`, and `C` as appropriate.

```scala
sealed trait MacbookPro
final case class ThirteenInch() extends MacbookPro
final case class FourteenInch() extends MacbookPro
final case class SixteenInch() extends MacbookPro
```

A 13" model has a specific choice of RAM **and** a specific choice of SSD. This is a logical and, meaning the translation to Scala looks like

```
final case class ThirteenInch(ram: Ram, ssd: Ssd)
```

where we assume an appropriate definition of the types `Ram` and `Ssd`.

Here's a complete definition of the model above.

```
sealed trait ThirteenInchRam
final case class EightGb() extends ThirteenInchRam
final case class SixteenGb() extends ThirteenInchRam

sealed trait ThirteenInchSsd
final case class TwoHundredAndFiftyGb() extends ThirteenInchSsd
final case class FiveHundredAndTwelveGb() extends ThirteenInchSsd
final case class OneTb() extends ThirteenInchSsd
final case class TwoTb() extends ThirteenInchSsd

sealed trait MacbookPro
final case class ThirteenInch(ram: ThirteenInchRam, ssd: ThirteenInchSsd)
    extends MacbookPro
final case class FourteenInch() extends MacbookPro
final case class SixteenInch() extends MacbookPro
```

To complete it we would need to define the options for the 14" and 16" models. Doing so doesn't illustrate anything we haven't already seen and makes the code much longer, so we'll skip that step.

Algebraic Data Types in Scala

There are some wrinkles to how algebraic data types are represented in Scala. The description above is sufficient for the beginner, but more advanced users should known the following.

Instead of a `sealed trait`, a `sealed abstract class` can be used. This is what you'll see in the standard library. For example, look at `Option`.

If a type holds no data, we can use a `case object` instead of a `final case class`. For example, none of the types that make up `ThirteenInchRam` or `ThirteenInchSsd` hold data, so we could instead write

```
sealed trait ThirteenInchRam
case object EightGb() extends ThirteenInchRam
case object SixteenGb() extends ThirteenInchRam

sealed trait ThirteenInchSsd
case object TwoHundredAndFiftyGb() extends ThirteenInchSsd
case object FiveHundredAndTwelveGb() extends ThirteenInchSsd
case object OneTb() extends ThirteenInchSsd
case object TwoTb() extends ThirteenInchSsd
```

We don't need the `final` modifier on a `case object` as objects cannot be extended.

It is often the case that we don't expect end users to work directly with the leaves of a sum type (a logical or). In this case it is conventional to put them in the companion object of the overall type. For example, imagine that we didn't want users to work directly with the subtypes of `ThirteenInchRam`. We would then, by convention, write

```
sealed trait ThirteenInchRam
object ThirteenInchRam {
  case object EightGb() extends ThirteenInchRam
  case object SixteenGb() extends ThirteenInchRam
}
```

Structural Recursion

To be continued…