Notes — Algebraic Data Types

Software Design and Programming/Software Programming III

Spring term 2017

In these notes we are going to look at modelling data and learn a process for expressing in Scala any data model defined in terms of logical *or*s and *and*s. Using the terminology of object-oriented programming, we will express *is-a* and *has-a* relationships. In the terminology of functional programming, we are learning about *sum* and *product* types, which are together called **algebraic data types**, not to be confused with *abstract data types*.

Our aims for these notes are:

- to see how to translate a data model into Scala code, and
- to examine patterns for code that uses algebraic data types.

The Product Type Pattern

Our first pattern is to model data that contains other data. We might describe this as

> "A has a B and C".

For example, a Cat has a colour and a favourite food; a Visitor has an id and a creation date; and so on.

The way we write this is to use a *case class*. We've already done this many several times in the exercises but now we are formalising the pattern.

If A has a b (with type B) and a c (with type C) write

```
case class A(b: B, c: C)
```

or

```
trait A {
        def b: B
        def c: C
}
```

The Sum Type Pattern

Our next pattern is to model data that is two or more distinct cases. We might describe this as

> "A is a B or C"

For example, a Feline is a Cat, Lion, or Tiger; a Visitor is an Anonymous or User; and so on.

We can represent this in Scala using the sealed trait/final case class pattern.

If A isa B or C write

```
sealed trait A
final case class B() extends A
final case class C() extends A
```

Algebraic Data Types

An algebraic data type is any data that uses the above two patterns. In the functional programming literature, data using the "has-a and" pattern is known as a *product type*, and the "is-a or" pattern is a *sum type*.

The Missing Patterns

We have looked at relationships along two dimensions: is-a/has-a, and and/or. We can draw up a table and see we only have patterns for two of the four table cells.

	AND	OR
Is-a	???	Sum type
Has-a	Product type	???

What about the missing two patterns?

The "is-a and" pattern means that A is a B and C. This pattern is in some ways the inverse of the sum type pattern, and we can implement it as

```
trait B
trait C
trait A extends B with C
```

In Scala a trait can extend as many traits as we like using the `with` keyword, e.g., `A extends B with C with D` and so on. If we want to represent that some data conforms to a number of different interfaces we will often be better off using a *type class*, more of which later. There are, however, several legitimate uses of this pattern:

- for modularity, using what is known as the *cake pattern*, and
- for sharing implementation across several classes where it doesn't make sense to make default implementations in the main trait.

The "has-a or" patterns means that A has a B or C. There are two ways we can implement this. We can say that A has a d of type D, where D is a B or C. We can mechanically apply our two patterns to implement this:

```
trait A {
        def d: D
}

sealed trait D
final case class B() extends D
final case class C() extends D
```

Alternatively we could implement this as A is a D or E, and D has a B and E has a C. Again this translates directly into code:

```
sealed trait A
final case class D(b: B) extends A
final case class E(c: C) extends A
```

Summary

We can mechanically translate data using the "has-a and" and "is-a or" patterns (or, more succinctly, the product and sum types) into Scala code. This type of data is known as an algebraic data type. Understanding these patterns is very important for writing idiomatic Scala code.

Structural Recursion

We have now seen how to define algebraic data types using a combination of the sum (or) and product type (and) patterns. In these notes we will examine a pattern for using algebraic data types, known as *structural recursion*. We will actually examine two variants of this pattern:

- one using polymorphism, and
- one using pattern matching.

Structural recursion is the precise opposite of the process of building an algebraic data type. If A has a B and C (the product-type pattern), to construct an A we must have a B and a C. The sum and product type patterns tell us how to combine data to make bigger data. Structural recursion says that if we have an A as defined before, we must break it into its constituent B and C that we then combine in some way to get closer to our desired answer.

Structural recursion is essentially the process of breaking down data into smaller pieces. Just as we have two patterns for building algebraic data types, we will have two patterns for decomposing them using structural recursion. We will actually have two variants of each pattern:

- one using polymorphism, which is the typical object-oriented style, and
- one using pattern matching, which is typical functional style.

We will end these notes with some rules for choosing which pattern to use and when.

Structural Recursion using Polymorphism

Polymorphic dispatch, or just polymorphism for short, is a fundamental object-oriented technique. If we define a method in a trait, and have different implementations in classes extending that trait, when we call that method the implementation on the actual concrete instance will be used.

Here's a very simple example. We start with a simple definition using the familiar sum type (or) pattern.

```
sealed trait A {
  def foo: String
}

final case class B() extends A {
  def foo: String = "It's B!"
}

final case class C() extends A {
  def foo: String = "It's C!"
}
```

We declare a value with type `A` but we see the concrete implementation on `B` or `C` is used.

```
scala> val anA: A = B()
anA: A = B()

scala> anA.foo
res1: String = It's B!
```

```
scala> val anA: A = C()
anA: A = C()

scala> anA.foo
res2: String = It's C!
```

We can define an implementation in a trait, and change the implementation in an extending class using the `override` keyword.

```
sealed trait A {
  def foo: String =
    "It's A!"
}

final case class B() extends A {
  override def foo: String = "It's B!"
}

final case class C() extends A {
  override def foo: String = "It's C!"
}
```

The behaviour is as before; the implementation on the concrete class is selected.

```
scala> val anA: A = B()
anA: A = B()

scala> anA.foo
res3: String = It's B!
```

Remember that if you provide a default implementation in a trait, you should ensure that implementation is valid for all subtypes.

So how do we use polymorphism with an algebraic data types? We have actually seen everything we need, but let's make it explicit and see the patterns.

The Product Type Polymorphism Pattern

If A has a b (with type B) and a c (with type C), and we want to write a method f returning an F, simply write the method in the usual way:

```
case class A(b: B, c: C) {
        def f: F = ???
}
```

In the body of the method we must use b, c, and any method parameters to construct the result of type F.

The Sum Type Polymorphism Pattern

If Ai s a B or C, and we want to write a method f returning an F, define f as an abstract method on A and provide concrete implementations in B and C.

```
sealed trait A {
  def f: F
}

final case class B() extends A {
  def f: F = ???
}

final case class C() extends A {
  def f: F = ???
}

trait F
```

Structural Recursion using Pattern Matching

Structural recursion with pattern matching proceeds along the same lines as polymorphism. We simply have a case for every subtype, and each pattern matching case must extract the fields we're interested in.

The Product Type Pattern Matching Pattern

If A has a b (with type B) and a c (with type C), and we want to write a method f that accepts an A and returns an F, write:

```
def f(a: A): F =
        a match {
                case A(b, c) => ???
        }
```

In the body of the method we use b and c to construct the result of type F.

The Sum Type Pattern Matching Pattern

If A is a B or C, and we want to write a method f accepting an A and returning an F, define a pattern matching case for B and C.

```
def f(a: A): F =
        a match {
                case B() => ???
                case C() => ???
        }
```

1 An example

Let us consider a complete example of the algebraic data type and structural recursion patterns; we will use a Feline data type describing cat-like creatures.

We start with a description of the data. A Feline is a Lion, Tiger, Panther, or Cat. We're going to simplify the data description, and just say that a Cat has a String `favouriteFood`. From this description we can immediately apply our pattern to define the data.

```
package example

sealed trait Feline

final case class Lion() extends Feline

final case class Tiger() extends Feline

final case class Panther() extends Feline

final case class Cat(favouriteFood: String) extends Feline
```

Now let's implement a method using both polymorphism and pattern matching. Our method, dinner, will return the appropriate food for the feline in question. For a Cat their dinner is their `favouriteFood`. For Lions it is antelope, for Tigers it is Tiger food, and for Panthers it is Humans.

We could represent food as a String, but we can do better and represent it with a type. This avoids, for example, spelling mistakes in our code. So let's define our Food type using the now familiar patterns:

```
package example

sealed trait Food

final case object Antelope extends Food

final case object TigerFood extends Food

final case object Humans extends Food

final case class CatFood(food: String) extends Food
```

Now we can implement dinner as a method returning Food. First using polymorphism:

```
package examplepoly

import example._

sealed trait Feline {
  def dinner: Food
}

final case class Lion() extends Feline {
  def dinner: Food = Antelope
}

final case class Tiger() extends Feline {
  def dinner: Food = TigerFood
}

final case class Panther() extends Feline {
  def dinner: Food = Humans
}

final case class Cat(favouriteFood: String) extends Feline {
  def dinner: Food = CatFood(favouriteFood)
}
```

Now using pattern matching. We actually have two choices when using pattern matching. We can implement our code in a single method on Feline or we can implement it in a method on another object. Let's see both:

```
package examplepm

import example.{Antelope, CatFood, Food, Humans, TigerFood}

sealed trait Feline {
  def dinner: Food =
    this match {
      case Lion() => Antelope
      case Tiger() => TigerFood
      case Panther() => Humans
      case Cat(favouriteFood) => CatFood(favouriteFood)
    }
}

object Diner {
  def dinner(feline: Feline): Food =
    feline match {
      case Lion() => Antelope
      case Tiger() => TigerFood
      case Panther() => Humans
      case Cat(food) => CatFood(food)
    }
}

final case class Lion() extends Feline

final case class Tiger() extends Feline

final case class Panther() extends Feline

final case class Cat(favouriteFood: String) extends Feline
```

Note how we can directly apply the patterns, and the code falls out. This is the main point we want to make with structural recursion: the code follows the shape of the data, and can be produced in an almost mechanical way.

Choosing Which Pattern to Use

We have three way of implementing structural recursion:

1. polymorphism;
2. pattern matching in the base trait, and
3. pattern matching in an external object (as in the `Diner` example above).

Which should we use? The first two methods give the same result: a method defined on the classes of interest. We should use whichever is more convenient. This normally ends up being pattern matching on the base trait as it requires less code duplication.

When we implement a method in the classes of interest we can have only one implementation of the method, and everything that method requires to work must be contained within the class and parameters we pass to the method. When we implement methods using pattern matching in an external object we can provide multiple implementations, one per object (multiple `Diner`s in the example above).

The general rule is:

- If a method only depends on other fields and methods in a class it is a good candidate to be implemented inside the class.
- If the method depends on other data (for example, if we needed a `Cook` to make dinner) consider implementing is using pattern matching outside of the classes in question.
- If we want to have more than one implementation we should use pattern matching and implement it outside the classes.

Object-Oriented vs Functional Extensibility

In classic functional programming style we have no objects, only data without methods and functions. This style of programming makes extensive use of pattern matching. We can mimic it in Scala using the algebraic data type pattern and pattern matching in methods defined on external objects.

Classic object oriented style uses polymorphism and allow open extension of classes. In Scala terms this means no sealed traits.

What are the tradeoffs we make in the two different styles?

One advantage of functional style is it allows the compiler to help us more. By sealing traits we are telling the compiler it knows all the possible subtypes of that trait. It can then tell us if we miss out a case in our pattern matching. This is especially useful if we add or remove subtypes later in development. We could argue we get the same benefit from object-oriented style, as we must implement all methods defined on the base trait in any subtypes. This is true, but in practice classes with a large number of methods are very difficult to maintain and we'll inevitably end up factoring some of the code into different classes — essentially duplicating the functional style.

This doesn't mean functional style is to be preferred in all cases. There is a fundamental difference between the kind of extensibility that object-oriented style and functional style gives us. With OO style we can easily add new data, by extending a trait, but adding a new method requires us to change existing code. With functional style we can easily add a new method but adding new data requires us to modify existing code. In tabular form:

	add new method	add new data
OO	Change existing code	Existing code unchanged
FP	Existing code unchanged	Change existing code

In Scala we have the flexibility to use both polymorphism and pattern matching, and we should use whichever is appropriate. However we generally prefer sealed traits as it gives us greater guarantees about our code's semantics, and we can use *typeclasses*, which we'll discuss later in the course, to achieve OO-style extensibility.

Acknowledgements

Thanks to Noel Welsh for the basis of these notes.