

Working With Algebraic Data Types



What are Algebraic Data Types?

- An approach to defining types that model certain patterns of data
 - Different to OO approach
 - Well suited to functional programming
 - Product types
 - Based on the "has-a and" relationship
 - *A has a B and a C*
 - Sum types
 - Based on "is-a or" relationship
 - *A is a B or a C*
 - Scala provides support for both of these
-

Product Types

- Models the "has-a and" relationship
- Implement using case classes
 - Cartesian product

```
case class Trade ( id: String,           // A trade has an id
                  side: String,         // and a side
                  sym: String,          // and a symbol
                  amount: Int,          // and an amount
                  unitPrice: Double     // and a unitPrice
                  )
```

```
case class Person ( fName: String,      // A person has a first name
                   lName: String,       // and a last name
                   age: Int             // and an age
                   )
```

Sum Types

- Models the "is-a or" relationship
- Implement using sealed traits and subtypes

```
sealed trait User                // A user
final case class Normal() extends User // is a normal user
final case class Privileged() extends User // or privileged
final case class Admin() extends User   // or an admin user
```

```
sealed trait Account            // An account
final case class Current() extends Account // is current
final case class Savings() extends User   // or savings
final case class Investment() extends User // or investment
```

Working With ADTs

- Code structure follows data structure

- Product types straightforward
- Scala.Product trait

- Sum types offer a choice

- Traditional OO polymorphism?

```
sealed trait Account {  
  def accountNo: String  
}  
final case class Current( accno: String ) extends Account {  
  def accountNo = s"C${accno}"  
}  
final case class Savings ( accno: String ) extends Account {  
  def accountNo = s"S${accno}"  
}  
final case class Investment ( accno: String ) extends Account {  
  def accountNo = s"I${accno}"  
}
```

Working With ADTs

- Alternative approach to working with Sum Types is Structural Recursion
 - Implemented using pattern matching

```
sealed trait Account {  
  def accountNo = this match {  
    case Current(anum)    => s"C${anum}"  
    case Savings(anum)    => s"S${anum}"  
    case Investment(anum) => s"I${anum}"  }  
}  
  
final case class Current ( accno: String ) extends Account  
final case class Savings ( accno: String ) extends Account  
final case class Investment ( accno: String ) extends Account
```

OO Polymorphism or Structural Recursion?

- Different approaches suit different situations
 - OO approach favours polymorphism
 - Functional approach favours structural recursion

	Add a new method	Add new data
OO	Change existing code	No change to existing code
FP	No change to existing code	Change existing code

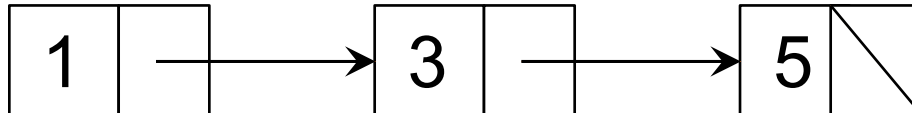
- OO approach handles data extensibility slightly better
 - New data types have localised functionality
 - Functional approach handles functionality extensibility better
 - New functionality added in base type only
 - Scala supports both approaches
-

Recursive Types

- Many types can be defined as recursive sum types
 - Eg List of Int values

```
sealed trait IntList                                // An IntList
final case object End extends IntList               // is empty
final case class Cons ( head: Int,                  // or an Int,
                        tail: IntList )             // followed by an IntList
                        extends IntList
```

```
scala> val l1: IntList = Cons ( 1, Cons(3, Cons(5, End) ) )
l1: IntList = Cons(1,Cons(3,Cons(5,End)))
```



Recursive Types

- Recursive sum types can be processed using structural recursion

```
def tail ( l: IntList ): IntList =  
  l match {  
    case End => End  
    case Cons( head, tail ) => tail  
  }
```

```
def length ( l: IntList ): Int =  
  l match {  
    case End => 0  
    case Cons(head, tail) => 1 + length(tail)  
  }
```

```
scala> length(l1)  
res3: Int = 3
```

```
scala> tail(l1)  
res4: IntList = Cons(3,Cons(5,End))
```

Recursive Types

- Different operations follow a similar pattern

```
def sum ( l: IntList ): Int =  
  l match {  
    case End => 0  
    case Cons(head, tail) => head + sum(tail)  
  }
```

```
def square( l: IntList ): IntList =  
  l match {  
    case End => End  
    case Cons(head, tail) => Cons(head * head, square(tail))  
  }
```

Base Case

Recursive Case

```
scala> sum(l1)  
res6: Int = 9
```

```
scala> square(l1)  
res5: IntList = Cons(1,Cons(9,Cons(25,End)))
```

Recursive Types

- Add operations to the type
 - FP approach

```
sealed trait IntList {  
  def length: Int = this match {  
    case End => 0  
    case Cons(hd, tl) => 1 + tl.length  
  }  
  def sum: Int = this match {  
    case End => 0  
    case Cons(hd, tl) => hd + tl.sum  
  }  
}
```

```
final case object End extends IntList  
final case class Cons(hd: Int, tl: IntList) extends IntList
```

```
scala> val l1: IntList =  
        Cons(1, Cons(3, Cons(5, End) ) )  
l1: IntList = Cons(1,Cons(3,Cons(5,End)))  
  
scala> l1.sum  
res85: Int = 9
```

Improving the Method Definitions

- Abstract over repeated code

```
sealed trait IntList {  
  def process (baseCase: Int, f: (Int, Int) => Int ): Int =  
    this match {  
      case End => baseCase  
      case Cons(hd, tl) => f( hd, tl.process(baseCase, f) )  
    }  
  
}  
  
final case object End extends IntList  
final case class Cons(hd: Int, tl: IntList) extends IntList
```

Improving the Method Definitions

- Abstract over repeated code

```
sealed trait IntList {  
  def process (baseCase: Int, f: (Int, Int) => Int ): Int =  
    this match {  
      case End => baseCase  
      case Cons(hd, tl) => f( hd, tl.process(baseCase, f) )  
    }  
  def sum: Int = process(0, (hd, tl) => hd + tl)  
  def length: Int = process(0, (_, tl) => 1 + tl)  
}  
  
final case object End extends IntList  
final case class Cons(hd: Int, tl: IntList) extends IntList
```

Improving the Method Definitions

- Abstract over repeated code

```
sealed trait IntList {  
  def fold (baseCase: Int, f: (Int, Int) => Int ): Int =  
    this match {  
      case End => baseCase  
      case Cons(hd, tl) => f( hd, tl.process(baseCase, f) )  
    }  
  def sum: Int = process(0, (hd, tl) => hd + tl)  
  def length: Int = process(0, (_, tl) => 1 + tl)  
}  
  
final case object End extends IntList  
final case class Cons(hd: Int, tl: IntList) extends IntList
```

Generalising the fold Method

- Currently we can only use fold for sum and length
 - Return type fixed as Int
- The square method cannot be implemented in this way
 - Return type is IntList
- Add type parameter to method definition


```
...  
def fold[A] (baseCase: A, f:(Int, A) => A): A =  
  this match {  
    case End => baseCase  
    case Cons(hd, tl) => f( hd, tl.fold(baseCase, f) )  
  }  
def sum: Int = fold[Int](0, (hd, tl) => hd + tl)  
def square: IntList = fold[IntList](End, (hd, tl) => Cons(hd*hd, tl) )  
...
```

Generic Algebraic Data Types

- ADTs can be parameterised like any other type

```
sealed trait MyList[A] {  
  ...  
}
```

Must be case class
as can't parameterise
an object



```
case class End[A]() extends MyList[A]  
case class Cons[A](hd: A, tl: MyList[A]) extends MyList[A]
```

```
scala> val ml1: MyList[String] = Cons("Hello", Cons("world", End()))  
ml1: MyList[String] = Cons(Hello,Cons(world,End()))
```

```
scala> val ml2: MyList[Int] = Cons(2, Cons(4, Cons(6, End() ) ) )  
ml2: MyList[Int] = Cons(2,Cons(4,Cons(6,End())))
```

Fold for Generic ADT

- Possible to generalise fold method for generic ADTs
 - Not all methods can be implemented
 - Length works as always returns Int

```
sealed trait MyList[A] {  
  
  def fold[B] ( end: B, f: (A, B) => B ): B =  
    this match {  
      case End() => end  
      case Cons(hd, tl) => f( hd, tl.fold(end, f) )  
    }  
  
  def length = fold [Int]( 0, (_, b) => 1 + b )  
}  
  
...
```

```
scala> ml1.length  
res89: Int = 2
```

```
scala> ml2.length  
res90: Int = 3
```

Fold for Generic ADT

- We can use fold directly to implement other functionality

```
scala> ml2.fold[Int] (0, (a,b) => a + b)
res92: Int = 12

scala> ml1.fold[String] ( "", (a,b) => s"$a $b" )
res93: String = "Hello world "
```

- Notice we must explicitly instantiate the type parameter
 - Type inference in the compiler unable to work out correct type
-

Fold for Generic ADT

- It is possible to help type inference
 - Use multiple argument lists in function

```
...  
def fold[B] (end: B) (f: (A, B) => B ): B =  
  this match {  
    case End() => end  
    case Cons(hd, tl) => f( hd, tl.fold(end)(f) )  
  }  
  
def length = fold [Int](0) ((_, b) => 1 + b )  
...
```

```
scala> m11.fold("")( (a,b) => s"$a $b" )  
res97: String = "Hello world "
```

```
scala> m12.fold(0)( (a,b) => a + b )  
res98: Int = 12
```

Generic ADTs as Containers

- ADT can be used to represent a simple container type

```
sealed trait Box[A]  
  
final case class Empty[A]() extends Box[A]  
final case class Full[A](value: A) extends Box[A]
```

- A Box is either
 - Empty
 - Full, containing a value of type A
 - Like type Option[T]
 - What does fold mean for such a type?
-

Using fold on Generic Container ADT

- Structural recursion can be applied to container types
 - Fold method will still work

```
scala> val emptyBox: Box[Int] = Empty()  
emptyBox: Box[Int] = Empty()
```

```
scala> val fullBox: Box[Int] = Full(10)  
fullBox: Box[Int] = Full(10)
```

```
scala> emptyBox.fold(-1)(i => i)  
res100: Int = -1
```

```
scala> fullBox.fold(-1)(i => i)  
res101: Int = 10
```

```
scala> emptyBox.fold(-1)(i => i + 3 )  
res104: Int = -1
```

```
scala> fullBox.fold(-1)(i => i + 3)  
res102: Int = 13
```
