Algebraic Data Types

Mark Hibberd

Mar 28, 2011

Outline

- Gentle(ish) introduction to Algebraic Data Types (ADTs).
- Demonstrate patterns, or encodings, for reprsenting ADTs.
- Examples, using Haskell, Scala and Java.
- Trade-offs, and ways of dealing with them.
- Designing an algebraric data type.

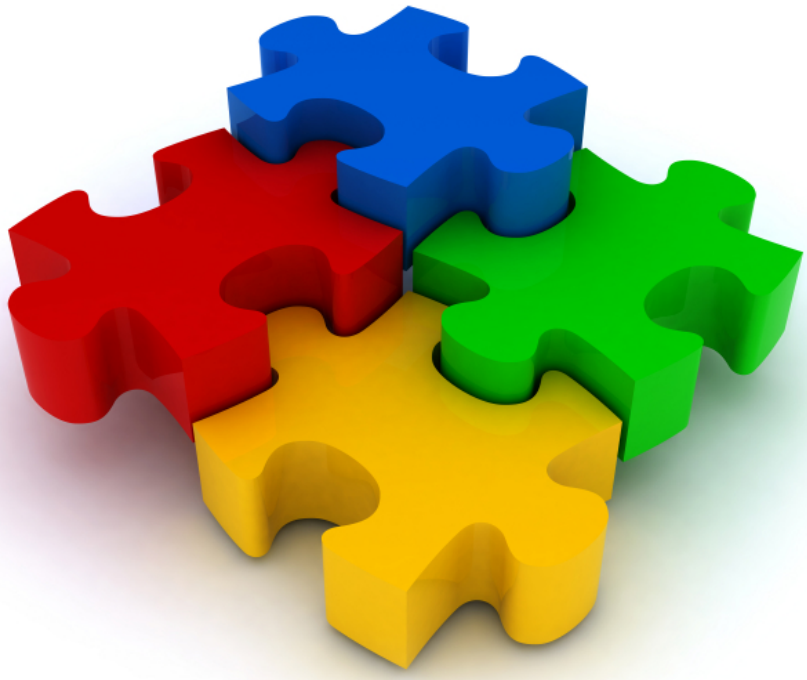
Terminology Overload

- Sum type
- Product type
- Disjoint, or tagged, union
- Variant type
- Recursive, or inductive, data type
- Enumeratated type
- Composite type
- Type constructor
- Data constructor
- Class constructor

Algebraic Data Types

- **Composite**. Definition by cases. Each case is composed into a single type.
- **Closed**. A finite set of cases.
- **Safe**. Provides mechanism for helping with (and enforcing) correct handling of all cases.
- **Prevalant**. The types of data we use every day.
- **Generalizable?**. There are a number of reliable patterns for defining combinators on algraic data-types.

Encoding Algebraic Data Types



Boolean: Haskell

```haskell
data Boolean = True | False
```

Boolean: Haskell

```haskell
1   data Boolean = True | False
```

- Type constructor - `Boolean`
- Two data constructors - `True` and `False`
- Disjoint union - *exactly one* of `True` or `False`
- Enumerated type - data constructors have no arguments.

Boolean: Scala

```
1  sealed trait Boolean
2  case object True extends Boolean
3  case object False extends Boolean
```

Boolean: Scala

```scala
sealed trait Boolean
case object True extends Boolean
case object False extends Boolean
```

```haskell
data Boolean = True | False
```

Church Encoding

- Representing data types (and their) operations in λ calculus.
- More simply, can we represent values as functions?
- Why is this useful?

Church Booleans

Formal Definition

$$true \equiv \lambda a.\lambda b.\ a$$
$$false \equiv \lambda a.\lambda b.\ b$$

Church Booleans: Scala

```
1  sealed trait Boolean {
2    def fold[A](t: => A, f: => A): A
3  }
```

Church Booleans: Scala

```
1   sealed trait Boolean {
2     def fold[A](t: => A, f: => A): A
3   }
4
5   object Boolean {
6     def True: Boolean = new Boolean {
7       def fold[A](t: => A, f: => A) = t
8     }
9
10    def False: Boolean = new Boolean {
11      def fold[A](t: => A, f: => A) = f
12    }
13  }
```

Church Booleans: Scala

```
1   sealed trait Boolean {
2     def fold[A](t: => A, f: => A): A
3   }
4
5   object Boolean {
6     def True: Boolean = new Boolean {
7       def fold[A](t: => A, f: => A) = t
8     }
9
10    def False: Boolean = new Boolean {
11      def fold[A](t: => A, f: => A) = f
12    }
13  }
```

$$true \equiv \lambda a.\lambda b.\, a$$
$$false \equiv \lambda a.\lambda b.\, b$$

Fold

- Fold is a catamorphism.
- A catamorphism is a higher order function that *defines* a data type.
- A catamorphism has the form:
  $A* \rightarrow B$
  where $A*$ is an algebraic data type, and B, is any other type.
- **All** functions on a type can be defined in terms of the catamorphism.
- If/Else is the catamorphism for boolean.

Church Boolean Operations: Scala

```scala
sealed trait Boolean {
  def fold[A](t: => A, f: => A): A

  def and(b: Boolean) = fold(b, m)

  def or(b: Boolean) = fold(this, b)

  def not = fold(False, True)

  def xor(b: Boolean) = fold(
    b.fold(False, True),
    b.fold(True, False)
  )
}
```

Church Boolean Operations: Scala

```scala
sealed trait Boolean {
  def fold[A](t: => A, f: => A): A

  def and(b: Boolean) = fold(b, m)

  def or(b: Boolean) = fold(this, b)

  def not = fold(False, True)

  def xor(b: Boolean) = fold(
    b.fold(False, True),
    b.fold(True, False)
  )
}
```

$$and \equiv \lambda m.\lambda n.\, m\, n\, m$$

Church Boolean Operations: Scala

```scala
sealed trait Boolean {
  def fold[A](t: => A, f: => A): A

  def and(b: Boolean) = fold(b, m)

  def or(b: Boolean) = fold(this, b)

  def not = fold(False, True)

  def xor(b: Boolean) = fold(
    b.fold(False, True),
    b.fold(True, False)
  )
}
```

$$or \equiv \lambda m.\lambda n.\ m\ m\ n$$

Church Boolean Operations: Scala

```scala
sealed trait Boolean {
  def fold[A](t: => A, f: => A): A

  def and(b: Boolean) = fold(b, m)

  def or(b: Boolean) = fold(this, b)

  def not = fold(False, True)

  def xor(b: Boolean) = fold(
    b.fold(False, True),
    b.fold(True, False)
  )
}
```

$$not \equiv \lambda m.\lambda a.\lambda b.\ m\ b\ a$$

Church Boolean Operations: Scala

```scala
sealed trait Boolean {
  def fold[A](t: => A, f: => A): A

  def and(b: Boolean) = fold(b, m)

  def or(b: Boolean) = fold(this, b)

  def not = fold(False, True)

  def xor(b: Boolean) = fold(
    b.fold(False, True),
    b.fold(True, False)
  )
}
```

$$xor \equiv \lambda m.\lambda n.\lambda a.\lambda b.\ m\,(n\,b\,a)\,(n\,a\,b)$$

There is a machine in your type

Side bar

There is a striking similarity between the catamorphism of a datastructure, $A* \longrightarrow B$, and the type of a program, or even a virtual machine.

Church Booleans: Haskell

```
1  data Boolean = B {
2    fold :: forall a. a -> a -> a
3  }
4
5  true = B (\a _ -> a)
6  false = B (\_ b -> b)
7
8  and m n = fold m n m
9  or m n = fold m m n
10 not m = fold m true false
11 xor m n = fold m
12     (fold n false true)
13     (fold n false true)
```

$$true \equiv \lambda a.\lambda b.\, a$$
$$false \equiv \lambda a.\lambda b.\, b$$

Church Booleans: Haskell

```
1  data Boolean = B {
2    fold :: forall a. a -> a -> a
3  }
4
5  true = B (\a _ -> a)
6  false = B (\_ b -> b)
7
8  and m n = fold m n m
9  or m n = fold m m n
10 not m = fold m true false
11 xor m n = fold m
12     (fold n false true)
13     (fold n false true)
```

$$and \equiv \lambda m.\lambda n.\, m\, n\, m$$

Church Booleans: Haskell

```haskell
data Boolean = B {
  fold :: forall a. a -> a -> a
}

true = B (\a _ -> a)
false = B (\_ b -> b)

and m n = fold m n m
or m n = fold m m n
not m = fold m true false
xor m n = fold m
    (fold n false true)
    (fold n false true)
```

$or \equiv \lambda m.\lambda n.\ m\ m\ n$

Church Booleans: Haskell

```
1  data Boolean = B {
2    fold :: forall a. a -> a -> a
3  }
4
5  true = B (\a _ -> a)
6  false = B (\_ b -> b)
7
8  and m n = fold m n m
9  or m n = fold m m n
10 not m = fold m true false
11 xor m n = fold m
12     (fold n false true)
13     (fold n false true)
```

$$not \equiv \lambda m.\lambda a.\lambda b.\, m\, b\, a$$

Church Booleans: Haskell

```
1  data Boolean = B {
2    fold :: forall a. a -> a -> a
3  }
4
5  true = B (\a _ -> a)
6  false = B (\_ b -> b)
7
8  and m n = fold m n m
9  or m n = fold m m n
10 not m = fold m true false
11 xor m n = fold m
12     (fold n false true)
13     (fold n false true)
```

$$xor \equiv$$
$$\lambda m.\lambda n.\lambda a.\lambda b.$$
$$m\,(n\,b\,a)\,(n\,a\,b)$$

Church Booleans: Java

```
1  public abstract class Boolean {
2    private Boolean() {}
3
4    public abstract <X> X fold(Thunk<X> t, Thunk<X> f);
5
6    public static Boolean True = new Boolean() {
7      public <X> X fold(Thunk<X> t, Thunk<X> f) {
8        return t.apply();
9      }
10   };
11   public static Boolean False = new Boolean() {
12     public <X> X fold(Thunk<X> t, Thunk<X> f) {
13       return f.apply();
14     }
15   };
16 }
```

Church Booleans: Java

```
1 interface Thunk<A> {
2   A apply();
3 }
```

Option: Haskell

```
1   data Option a = None | Some a
```

Option: Scala

```
1    sealed trait Option[A]
2    case class None[A]() extends Option[A]
3    case class Some[A](a: A) extends Option[A]
```

Option: Scala - Church Encoding

```scala
sealed trait Option[A] {
  def fold[X](none: => X, some: A => X): X
}
object Option {
  def None[A]: Option[A] = new Option[A] {
    def fold[X](none: => X, some: A => X): X = none
  }
  def Some[A](a: A): Option[A] = new Option[A] {
    def fold[X](none: => X, some: A => X): X = some(a)
  }
}
```

Option: Java - Church Encoding

```
1  public abstract class Option<A> {
2    private Option() {}
3    public abstract <X> X fold(Thunk<X> none, F<A, X> some);
4    public static <A> Option<A> None() {
5      return new Option<A>() {
6        public <X> X fold(Thunk<X> none, F<A, X> some) {
7          return none.apply();
8        }
9      };
10   }
11   public static <A> Option<A> Some(final A a) {
12     return new Option<A>() {
13       public <X> X fold(Thunk<X> none, F<A, X> some) {
14         return some.apply(a);
15       }
16     };
17   }
```

Pattern Matching

- What?
  - Recognise values.
  - Bind names to values.
  - Break down values into parts.
- Why?
  - Programming by cases.
  - Compiler can help for algebraic data types.

Option Pattern Matching: Haskell

```haskell
isSome :: Option a -> Bool
isSome None = False
isSome Some _ = True

isNone :: Option a -> Bool
isNone = not isSome

map :: (a -> b) -> Option a -> Option b
map _ None = None
map f Some s = Some (f s)

toList :: Option a -> [a]
toList None = []
toList Some s = [a]
```

Option (Poor) Pattern Matching: Scala

```
1   val option = somePartialFunction(args)
2
3   val mapping = option match {
4     case None => None
5     case Some(s) => Some(needsAnS(s))
6   }
7
8   val set = option match {
9     case None => false
10    case Some(_) => true
11  }
12
13  val default = option match {
14    case None => defaultvalue
15    case Some(s) => s
16  }
```

Option More (Poor) Pattern Matching: Scala

```
1   val specialcase = option match {
2     case None => None
3     case Some(‘nugget‘) => None
4     case s@Some(_) => s
5   }
6
7   val guardcase = option match {
8     case None => None
9     case Some(s) if s == nugget => None
10    case s@Some(_) => s
11  }
12
13  val overlapIsSet = option match {
14    case None => false
15    case _ => true
16  }
```

Pattern Matching

- What?
  - Recognise values.
  - Bind names to values.
  - Break down values into parts.
- Why?
  - Programming by cases.
  - Compiler can help for algebraic data types.
- Why not?
  - Difficult to add a case for all operations.
  - Can replace all pattern matches with higher-order functions.

The Expression Problem

"The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)." – Philip Wadler, The Expression Problem, 1998

Expression Problem: Hard to add cases

```
1  data Shape = Circle Int | Square Int | Rectangle Int
       Int
2
3  area :: Shape -> Double
4  area Circle r = pi * r ^ 2
5  area Square s = s ^ 2
6  area Rectangle w h = w * h
```

Expression Problem: Hard to add cases

- Easy to add operations:

	Area	Perimeter
Circle		
Square		
Rectangle		

- Difficult to add cases:

	Area
Circle	
Square	
Rectangle	
Triangle	

Expression Problem: Hard to add cases

```
1   data Shape = Circle Int | Square Int | Rectangle Int
        Int
2
3   area :: Shape -> Double
4   area Circle r = pi * r ^ 2
5   area Square s = s ^ 2
6   area Rectangle w h = w * h
7
8   perimeter :: Shape -> Double
9   perimeter Circle r = 2 * pi * r
10  perimeter Square s = s * 4
11  perimeter Rectangle w h = (w + h) * 2
```

Expression Problem: Hard to add cases

```
1   data Shape = Circle Int | Square Int | Rectangle Int
        Int | Triangle Int Int Int
2
3   area :: Shape -> Double
4   area Circle r = pi * r ^ 2
5   area Square s = s ^ 2
6   area Rectangle w h = w * h
7   area Triangle o a h = o * a / 2
8
9   perimeter :: Shape -> Double
10  perimeter Circle r = 2 * pi * r
11  perimeter Square s = s * 4
12  perimeter Rectangle w h = (w + h) * 2
13  perimeter Triangle o a h = o + a + h
```

Expression Problem: Hard to add operations

```
1   trait Shape {
2     def area: Double
3   }
4   case class Circle(r: Int) extends Shape {
5     def area = Math.PI * Math.pow(r, 2)
6   }
7   case class Square(s: Int) extends Shape {
8     def area = Math.pow(s, 2)
9   }
10  case class Rectangle(w: Int, h: Int) extends Shape {
11    def area = w * h
12  }
```

Expression Problem: Hard to add operations

- Easy to add cases:

	Area
Circle	
Square	
Rectangle	
Triangle	

- Difficult to add operations:

	Area	Perimeter
Circle		
Square		
Rectangle		

Expression Problem: Hard to add operations

```
1   trait Shape {
2     def area: Double
3   }
4   case class Circle(r: Int) extends Shape {
5     def area = Math.PI * Math.pow(r, 2)
6   }
7   case class Square(s: Int) extends Shape {
8     def area = Math.pow(s, 2)
9   }
10  case class Rectangle(w: Int, h: Int) extends Shape {
11    def area = w * h
12  }
13  case class Triangle(o: Int, a: Int, h: Int) extends
        Shape {
14    def area = o * a / 2
15  }
```

Expression Problem: Hard to add operations

```
1   trait Shape {
2     def area: Double
3     def perimeter: Double
4   }
5   case class Circle(r: Int) extends Shape {
6     def area = Math.PI * Math.pow(r, 2)
7     def perimeter = 2 * Math.PI * r
8   }
9   case class Square(s: Int) extends Shape {
10    def area = Math.pow(s, 2)
11    def perimeter = s * 4
12  }
13  case class Rectangle(w: Int, h: Int) extends Shape {
14    def area = w * h
15    def perimeter = (w + h) * 2
16  }
```

Expression Problem: Hard to add operations

```
1   data Circle Int
2   data Square Int
3   data Rectangle Int Int
4
5   class Shape a where
6     area :: a -> Double
7
8   instance Shape Circle where
9     area Circle r = pi * r ^ 2
10  instance Shape Square where
11    area Square s = s ^ 2
12  instance Shape Rectangle where
13    area Rectangle w h = w * h
```

Expression Problem: Hard to add cases

```
1   trait Shape
2   case class Circle(r: Int) extends Shape
3   case class Square(s: Int) extends Shape
4   case class Rectangle(w: Int, h: Int) extends Shape
5
6   def area(s: Shape) = s match {
7     case Circle(r) => Math.PI * Math.pow(r, 2)
8     case Square(s) => Math.pow(s, 2)
9     case Rectangle(w, h) => w * h
10  }
```

Alleviating the Expression Problem with Combinators

Alleviating the Expression Problem with Combinators

```
1   sealed trait Option[A] {
2     def fold[X](none: => X, some: A => X): X
3   }
4   object Option {
5     def None[A]: Option[A] = new Option[A] {
6       def fold[X](none: => X, some: A => X): X = none
7     }
8     def Some[A](a: A): Option[A] = new Option[A] {
9       def fold[X](none: => X, some: A => X): X = some(a)
10    }
11  }
```

Alleviating the Expression Problem with Combinators

```
1   sealed trait Option[A] {
2     def fold[X](none: => X, some: A => X): X
3
4     def map[B](f: A => B) = fold(None, v => Some(f(v))
5
6     def flatMap[B](f: A => Option[B]) = fold(None, v =>
          f(v))
7
8     def orElse(other: => A) = fold(other, a => a)
9
10    def isSet = fold(false, true)
11
12    def isNotSet = !isSet
13  }
```

Alleviating the Expression Problem with Combinators

```
1    val mapping = option match {
2      case None => None
3      case Some(s) => Some(needsAnS(s))
4    }
```

Alleviating the Expression Problem with Combinators

```
1   val mapping = option match {
2     case None => None
3     case Some(s) => Some(needsAnS(s))
4   }
```

```
1   val mapping = option map (needsAnS(_))
```

Alleviating the Expression Problem with Combinators

```
1  val set = option match {
2    case None => false
3    case Some(_) => true
4  }
```

Alleviating the Expression Problem with Combinators

```
1  val set = option match {
2    case None => false
3    case Some(_) => true
4  }
```

```
1  option.isSet
```

Alleviating the Expression Problem with Combinators

```
1  val default = option match {
2    case None => defaultvalue
3    case Some(s) => s
4  }
```

Alleviating the Expression Problem with Combinators

```
1  val default = option match {
2    case None => defaultvalue
3    case Some(s) => s
4  }
```

```
1  option.orElse(defaultValue)
```

Alleviating the Expression Problem with Combinators

```
1   val chain = option match {
2     case None => defaultvalue
3     case Some(s1) => f1(s1) match {
4       case None => defaultvalue
5       case Some(s2) => f2(s2) match {
6         case None => defaultvalue
7         case Some(s3) => f3(s3) match {
8           case None => defaultvalue
9           case Some(s4) => s4
10        }
11      }
12    }
13  }
```

Alleviating the Expression Problem with Combinators

```
1  val s4 =
2    option.flatMap(
3     f1(_)).flatMap(
4     f2(_)).flatMap(
5     f3(_))
6  s4.orElse(defaultValue)
```

Alleviating the Expression Problem with Combinators

```
1  val s4 = for {
2    o <- option
3    s1 <- f1(s1)
4    s2 <- f2(s2)
5    s3 <- f3(s3)
6  } yield s3
7  s4.orElse(defaultValue)
```

Alleviating the Expression Problem with Combinators

```
1  import scalaz._, Scalaz._
2
3  val s4 = option >>= (f1(_)) >>= (f2(_)) >>= (f3(_))
4  s4.orElse(defaultValue)
```

Concept Summary

- We can use built in language constructs or church encoding to represent algebraic data types.
- Everything is defined in terms of fold, or the catamorphism, of some data structure.
- The expression problem is evident, no matter what approach, but we can help combat it by hiding constructors, using combinators and having good language support.

Questions

Designing an application around ADTs

- Accessing data from a database.
- A data type to represent a result.
- A data type to represent a query.
- A data type to represent an update.
- A data type to represent the execution of query or update to obtain a result.

Design

```
1  data DbValue a = Err String | Nul | Value a
2
3  data Variable = S String | I Int
4
5  data Ddl = D String [Variable]
6
7  data Query a = Q String [Variable] (ResultSet ->
       DbValue a)
8
9  data Connector a = C (Connection -> DbValue a)
```

To The Code

That's it.

References

1. Meijer, E. and Fokkinga, M. and Paterson, R. - Functional programming with bananas, lenses, envelopes and barbed wire - http://bit.ly/gOTczS
2. Philip Wadler - The Expression Problem - http://bit.ly/g86Guv
3. Runar Bjarnason - Functonal Programming for Beginners - http://vimeo.com/18554216 - http://bit.ly/gaEbxJ