Type classes and functional abstractions in Scala
A short and incomplete introduction

Jacek Malec

September 20, 2012

So: What are type classes?

*Typeclasses are among the **most powerful features** in Haskell. They allow you to define **generic interfaces** that provide a common feature set over a wide variety of types. Typeclasses are at the heart of some basic language features such as equality testing and numeric operators.*

(from OSG)

A motivating example

```
case class MyModel(val data: Int)

trait Ord[T] {
  def compare (x: T, y: T): Boolean
}
implicit object ordMyModel extends Ord[MyModel] {
  def compare (m1: MyModel, m2: MyModel) =
                                    m1.data <= m2.data
}
def choose[T](m1: T,m2: T)(implicit ordM: Ord[T]) =
  if (ordM.compare (m1,m2)) m2 else m1

val m1 = new MyModel(3)
val m2 = new MyModel(5)
choose(m1,m2)
```

Type classes

They allow us to

- introduce polymorphic functions, extensible (compositional) after the original code has been written (or even compiled)
- introduce generic functions in terms of the prototypes assumed to exist

Note use of the *implicit* mechanism to pass constraints around. Implicits are looked for in the local scope (contrary to Haskell).

On implicits

```
import java.io.PrintStream
implicit val out = System.out
def log (msg : String) (implicit o : PrintStream)
    = o.println (msg)

log ("Does not compute!")
log ("Does not compute!!")(System.err)

def logTm (msg :String)(implicit o :PrintStream):Unit
    = log ("[" + new java.util.Date () + "]" + msg)
```

More implicits?

```
def logPrefix (msg : String)
    (implicit o :PrintStream, prefix :String) :Unit
    = log ("["+prefix+"]"+msg)

//the look-up idiom
def?[T] (implicit w:T):T = w

//now we can say
logPrefix ("message") (?, "myprefix")
```

Scoping of implicits

```
trait Monoid [A ] {
  def binary_op (x:A,y:A):A
  def identity           :A
}

def acc[A] (l:List[A]) (implicit m:Monoid[A]):A =
  l.foldLeft(m.identity)((x, y) => m.binary_op(x, y))

object A {
  implicit object sumMonoid extends Monoid [Int ] {
    def binary_op (x:Int,y:Int) = x+y
    def identity = 0
  }
  def sum (l:List[Int]):Int = acc (l)
}
```

Scoping of implicits

```
object B {
  implicit object prodMonoid extends Monoid [Int ] {
    def binary_op (x:Int ,y:Int) = x*y
    def identity = 1
  }
  def product (l : List [Int ]) : Int = acc (l)
}

val test:(Int ,Int ,Int)= {
  import A._
  import B._
  val l = List (1 ,2 ,3 ,4 ,5)
  (sum (l), product (l), acc (l) (prodMonoid))
}
```

The CONCEPT pattern

Let's compare again:

```
trait Ord [T] {
  def compare (x:T,y:T) :Boolean
}
class Apple (x : Int) { }
object ordApple extends Ord [Apple ] {
  def compare (a1 :Apple,a2 :Apple) = a1.x < a2.x
}
def pick[T] (a1 :T,a2 :T) (ordA :Ord[T])
  = if (ordA.compare (a1,a2)) a2 else a1

val a1 = new Apple (3)
val a2 = new Apple (5)
val a3 = pick (a1,a2) (ordApple)
```

The CONCEPT pattern

Concepts are interfaces with generics.

The CONCEPT pattern

Concepts are interfaces with generics.
or
Concepts describe a set of requirements for the type parameters used by generic algorithms.

In our *Apple* example:

- trait `Ord[T]` is a *concept interface*
- `T` is the *modeled type*
- *Apple* is a *concrete modeled type*
- Actual objects implementing concept interfaces, such as `ordApple`, are called *models*

The CONCEPT pattern

The CONCEPT pattern can model n-ary, factory and consumer methods just like typeclasses.

We can model multi-type concepts:

```
trait Coerce[A,B] {
  def coerce (x :A) :B
}
```

zipWithN is another example (check the paper).

Benefits again:

- retroactive modeling
- multiple method implementations
- binary (or n-ary) methods
- factory methods

However: statically dispatched!

An alternative

Bounded polymorphism:

```
trait Ord[T] {
  def compare (x:T) :Boolean
}

class Apple (x:Int) extends Ord[Apple] ...
```

`compare` becomes a real, dynamically dispatched method of *Apple*. All the private info about apple objects is available for its definition. However, modeled types (*Apple*) have to state explicitely which concept interfaces they support: breaks retroactive modeling and multiple method implementations.

Abstract data types

Consider:

```
trait Set[S] {
  val empty : S
  def insert (x:S, y:Int)   :S
  def contains (x:S,y:Int) :Boolean
  def union (x:S,y:S)       :S
}
```

It may be considered to be an algebraic signature of an ADT *Set*.

Functional idioms and design patterns

- Immutable objects
- Higher order functions
  - taking functions as arguments
  - returning functions as results
  - possibly with their environment (closures)
- Lazy evaluation
  - encapsulated in objects
  - infinite data structures
- Clean separation of statefulness

Functional abstraction

Everything that uses `foreach` and `filter` (functionally better known as `map`, `flatMap` and `filter`) is usually *functional*

Functional abstraction

Everything that uses `foreach` and `filter` (functionally better known as `map`, `flatMap` and `filter`) is usually *functional*

and inherently *monadic* in nature.

Functional abstraction

Everything that uses `foreach` and `filter` (functionally better known as `map`, `flatMap` and `filter`) is usually *functional*

and inherently *monadic* in nature.

Implicits are crucial.

Functional abstraction

Everything that uses `foreach` and `filter` (functionally better known as `map`, `flatMap` and `filter`) is usually *functional*

and inherently *monadic* in nature.

Implicits are crucial.

Consider `scalaz` library, if you miss something you are used to.

References

- Björn
- The paper “Type classes as objects and implicits”, Bruno C.d.S. Oliveira, Adriaan Moors and Martin Odersky, OOPSLA 2010 (`http://ropas.snu.ac.kr/~bruno/papers/TypeClasses.pdf`)
- A site (`http://code.google.com/p/scalaz/`)
- The Scala textbook, 2nd ed.
- (OSG) A Haskell textbook (Bryan O’Sullivan, Don Stewart, and John Goerzen, `http://book.realworldhaskell.org/`)
- A blog page `http://www.codecommit.com/blog/ruby/monads-are-not-metaphors` by Daniel Spiewak.