Typeclasses in Scala

Grzegorz Balcerek

Scalania, 4 lutego 2015

collapse

```
scala> collapse(List("Hello"," ","World"))
res0: String = Hello World
```

collapse

```
scala> collapse(List("Hello"," ","World"))
res0: String = Hello World
```

```
scala> collapse(List(1,2,3,4))
res1: Int = 24
```

collapse

```
scala> collapse(List("Hello"," ","World"))
res0: String = Hello World
```

```
scala> collapse(List(1,2,3,4))
res1: Int = 24
```

```
val tree: Tree[Int] =
  Node(Node(Leaf,2,Leaf),4,Leaf)
```

```
scala> collapse(tree)
res2: Int = 8
```

collapse

collapse

```
def collapse(list: List[String]): String
def collapse(list: List[Int]): Int
def collapse(tree: Tree[Int]): Int
```

collapse

collapse

```
def collapse(list: List[String]): String
def collapse(list: List[Int]): Int
def collapse(tree: Tree[Int]): Int
```

Tree

```
trait Tree[+A]
case object Leaf extends Tree[Nothing]
case class Node[T](
  left: Tree[T],
  elem: T,
  right: Tree[T]) extends Tree[T]
```

collapse

```
def collapse(coll: C[T]): T
```

Abstractions

Abstractions

API

```
compare: (T,T) => Ordering
```

Abstractions

API

```
compare: (T,T) => Ordering
```

```
trait Ordering
case object LT extends Ordering
case object EQ extends Ordering
case object GT extends Ordering
```

Abstractions

API

```
compare: (T,T) => Ordering
```

```
trait Ordering
case object LT extends Ordering
case object EQ extends Ordering
case object GT extends Ordering
```

Laws

```
compare(a,b) = compare(b,c) = x => compare(a,c) = x
```

Abstractions

```
Int
```

Abstractions

```
Int
```

```
String
```

Abstractions

```
Int
```

```
String
```

```
Int => Int
```

Abstractions

```
sort: (List[T]) => List[T]
```

Comparator

```
trait Comparator[T] {
  def compare(a:T, b:T): Ordering
}
```

Comparator

```
trait Comparator[T] {
  def compare(a:T, b:T): Ordering
}
```

```
object IntComp extends Comparator[Int] {
  def compare(a:Int, b:Int) = ...
```

Comparator

```
trait Comparator[T] {
  def compare(a:T, b:T): Ordering
}
```

```
object IntComp extends Comparator[Int] {
  def compare(a:Int, b:Int) = ...
```

```
def sort[T](lst:List[T],
      c:Comparator[T]): List[T] = ...
```

Comparator

```
trait Comparator[T] {
  def compare(a:T, b:T): Ordering
}
```

```
object IntComp extends Comparator[Int] {
  def compare(a:Int, b:Int) = ...
```

```
def sort[T](lst:List[T],
      c:Comparator[T]): List[T] = ...
```

```
scala> sort(List(2,7,4,5,4,1), IntComp)
res0: List[Int] = List(1, 2, 4, 4, 5, 7)
```

Comparable

```
trait Comparable[T] {
  def compare(a:T): Ordering
}
```

Comparable

```
trait Comparable[T] {
  def compare(a:T): Ordering
}
```

```
case class ComparableRational(p:Int,q:Int = 1)
extends Comparable[ComparableRational] {
  def compare(x:ComparableRational): Ordering = ...
```

Comparable

```
trait Comparable[T] {
  def compare(a:T): Ordering
}
```

```
case class ComparableRational(p:Int,q:Int = 1)
extends Comparable[ComparableRational] {
  def compare(x:ComparableRational): Ordering = ...
```

```
def sort[T <: Comparable[T]](lst:List[T]): List[T] =
```

Comparable

```
trait Comparable[T] {
  def compare(a:T): Ordering
}
```

```
case class ComparableRational(p:Int,q:Int = 1)
extends Comparable[ComparableRational] {
  def compare(x:ComparableRational): Ordering = ...
```

```
def sort[T <: Comparable[T]](lst:List[T]): List[T] =
```

```
scala> sort(List(ComparableRational(3,2),
     |  ComparableRational(1,4), ComparableRational(1)))
res0: List[ComparableRational] = List(ComparableRational(1,4),
ComparableRational(1,1), ComparableRational(3,2))
```

Typeclasses

```
trait Comparator[T] {
  def compare(a:T, b:T): Ordering
}
```

Typeclasses

```
implicit object IntComparator
         extends Comparator[Int] {
  def compare(a:Int, b:Int) =
    if (a < b) LT
    else if (a > b) GT
    else EQ
}
```

Typeclasses

```
case class Rational(p:Int,q:Int = 1)
```

```
implicit object RationalComparator
         extends Comparator[Rational] {
  def compare(a:Rational,b:Rational): Ordering = {
     val r1 = a.p.toDouble/a.q.toDouble
     val r2 = b.p.toDouble/b.q.toDouble
     if (r1 < r2) LT else if (r1 > r2) GT else EQ
  }
}
```

Typeclasses

```
def sort[T](lst:List[T])
           (implicit c: Comparator[T]): List[T] =
  lst match {
    case Nil => Nil
    case h::_ =>
      sort(lst.filter(c.compare(h,_) == GT)) ++
      lst.filter(c.compare(h,_) == EQ) ++
      sort(lst.filter(c.compare(h,_) == LT))
  }
```

Typeclasses

```
scala> sort(List(2,7,4,5,4,1))
res0: List[Int] = List(1, 2, 4, 4, 5, 7)

scala> sort(List(Rational(3,2),
     |  Rational(1,4), Rational(1)))
res1: List[Rational] = List(Rational(1,4),
Rational(1,1), Rational(3,2))
```

Context Bound

Implicit parameters list

```
def sort[T](lst:List[T])
           (implicit c: Comparator[T]) =
...
```

Context Bound

Implicit parameters list

```
def sort[T](lst:List[T])
           (implicit c: Comparator[T]) =
...
```

Context Bound

```
def sort[T:Comparator](lst:List[T]) =
{
  val c = implicitly[Comparator[T]]
...
```

Multiple instances

```
implicit object StringComparator
         extends Comparator[String] {
  def compare(a:String, b:String) =
    if (a < b) LT
    else if (a > b) GT
    else EQ
}
```

Multiple instances

```
implicit object StringComparator
         extends Comparator[String] {
  def compare(a:String, b:String) =
    if (a < b) LT
    else if (a > b) GT
    else EQ
}
```

```
scala> sort(List("b","Z","A","x"))
res2: List[String] = List(A, Z, b, x)
```

Multiple instances

```
case class IgnoreCase(str: String) extends AnyVal
```

Multiple instances

```
case class IgnoreCase(str: String) extends AnyVal
```

```
implicit object IgnoreCaseStringComparator
         extends Comparator[IgnoreCase] {
  def compare(a:IgnoreCase, b:IgnoreCase) =
    if (a.str.toLowerCase < b.str.toLowerCase) LT
    else if (a.str.toLowerCase > b.str.toLowerCase) GT
    else EQ
}
```

Multiple instances

```
case class IgnoreCase(str: String) extends AnyVal
```

```
implicit object IgnoreCaseStringComparator
         extends Comparator[IgnoreCase] {
  def compare(a:IgnoreCase, b:IgnoreCase) =
    if (a.str.toLowerCase < b.str.toLowerCase) LT
    else if (a.str.toLowerCase > b.str.toLowerCase) GT
    else EQ
}
```

```
scala> sort(List("b","Z","A","x") map
     |  IgnoreCase.apply).map(_.str)
res3: List[String] = List(A, b, x, Z)
```

Multiple instances

```
trait NoCase
```

Multiple instances

```
trait NoCase
```

```
def tag[U](s:String): String with U =
  s.asInstanceOf[String with U]
```

Multiple instances

```
trait NoCase
```

```
def tag[U](s:String): String with U =
  s.asInstanceOf[String with U]
```

```
scala> tag[NoCase]("a")
res4: String with NoCase = a
```

Multiple instances

```
implicit object NoCaseStringComparator
         extends Comparator[String with NoCase] {
  def compare(a:String with NoCase,
              b:String with NoCase) =
    if (a.toLowerCase < b.toLowerCase) LT
    else if (a.toLowerCase > b.toLowerCase) GT
    else EQ
}
```

Multiple instances

```
implicit object NoCaseStringComparator
         extends Comparator[String with NoCase] {
  def compare(a:String with NoCase,
              b:String with NoCase) =
    if (a.toLowerCase < b.toLowerCase) LT
    else if (a.toLowerCase > b.toLowerCase) GT
    else EQ
}
```

```
scala> sort(List("b","Z","A","x") map tag[NoCase])
res5: List[String with NoCase] = List(A, b, x, Z)
```

collapse

```
def collapse(coll: C[T]): T
```

collapse

```
def collapse(coll: C[T]): T
```

- T — Monoid
- C — Foldable

Monoid

API

```
trait Monoid[T] {
  def mplus(a:T,b:T):T
  def mzero: T
}
```

Monoid

API

```
trait Monoid[T] {
  def mplus(a:T,b:T):T
  def mzero: T
}
```

Laws

```
mplus(a,mplus(b,c)) = mplus(mplus(a,b),c)
mplus(mzero,x) = x
mplus(x,mzero) = x
```

Monoid

```
implicit object StringMonoid
         extends Monoid[String] {
  def mplus(a: String, b: String) = a + b
  def mzero = ""
}
```

Monoid

```
implicit object StringMonoid
         extends Monoid[String] {
  def mplus(a: String, b: String) = a + b
  def mzero = ""
}
```

```
implicit object IntMonoid
         extends Monoid[Int] {
  def mplus(a: Int, b: Int) = a * b
  def mzero = 1
}
```

Foldable

```
import language.higherKinds
```

```
trait Foldable[F[_]] {
  def foldMap[A,B:Monoid](foldable: F[A])
                         (f: A => B): B
}
```

Foldable

```
implicit object FoldableList
         extends Foldable[List] {

  def foldMap[A,B](fa: List[A])(g: A => B)
                  (implicit m: Monoid[B]): B =
    fa.foldLeft(m.mzero)(
      (b,a) => m.mplus(b,g(a)) )

}
```

Foldable

```
implicit object FoldableTree
         extends Foldable[Tree] {
  def foldMap[A,B](fa: Tree[A])(g: A => B)
                  (implicit m: Monoid[B]): B =
    fa match {
      case Leaf => m.mzero
      case Node(left, elem, right) =>
        m.mplus(m.mplus(foldMap(left)(g),
                        g(elem)),
                foldMap(right)(g))
    }
}
```

collapse

```
def collapse[A:Monoid,F[A]:Foldable](f: F[A]): A =
  implicitly[Foldable[F]].foldMap(f)(identity)
```

collapse

```
scala> collapse(List("Hello"," ","World"))
res0: String = Hello World
```

collapse

```
scala> collapse(List("Hello"," ","World"))
res0: String = Hello World
```

```
scala> collapse(List(1,2,3,4))
res1: Int = 24
```

collapse

```
scala> collapse(List("Hello"," ","World"))
res0: String = Hello World
```

```
scala> collapse(List(1,2,3,4))
res1: Int = 24
```

```
val tree: Tree[Int] =
  Node(Node(Leaf,2,Leaf),4,Leaf)
```

```
scala> collapse(tree)
res2: Int = 8
```

Tuple2

Tuple2 instance

```
implicit def MonoidTuple2[S,T](implicit
                ms: Monoid[S], mt: Monoid[T]) =
  new Monoid[(S,T)] {
    def mzero = (ms.mzero,mt.mzero)
    def mplus(a: (S,T), b: (S,T)) =
      (ms.mplus(a._1, b._1),
       mt.mplus(a._2, b._2))
  }
```

Tuple2

```
scala> collapse(List(("Hello ",2),("World",4)))
res3: (String, Int) = (Hello World,8)
```

Tuple2

```
scala> collapse(List(("Hello ",2),("World",4)))
res3: (String, Int) = (Hello World,8)
```

```
val tree2: Tree[(String,Int)] = Node(
    Node(Leaf,("Hello ",2),Leaf),
    ("World",4),Leaf)
```

```
scala> collapse(tree2)
res4: (String, Int) = ("Hello World ",8)
```