Shapeless 101

Shapeless is an advanced functional programming library for the scala language.

Disclaimer

This talk is adressed to complete beginners in Shapeless.

Shapeless 101

Me : Harry Laoulakos (Software Engineer at Lunatech)

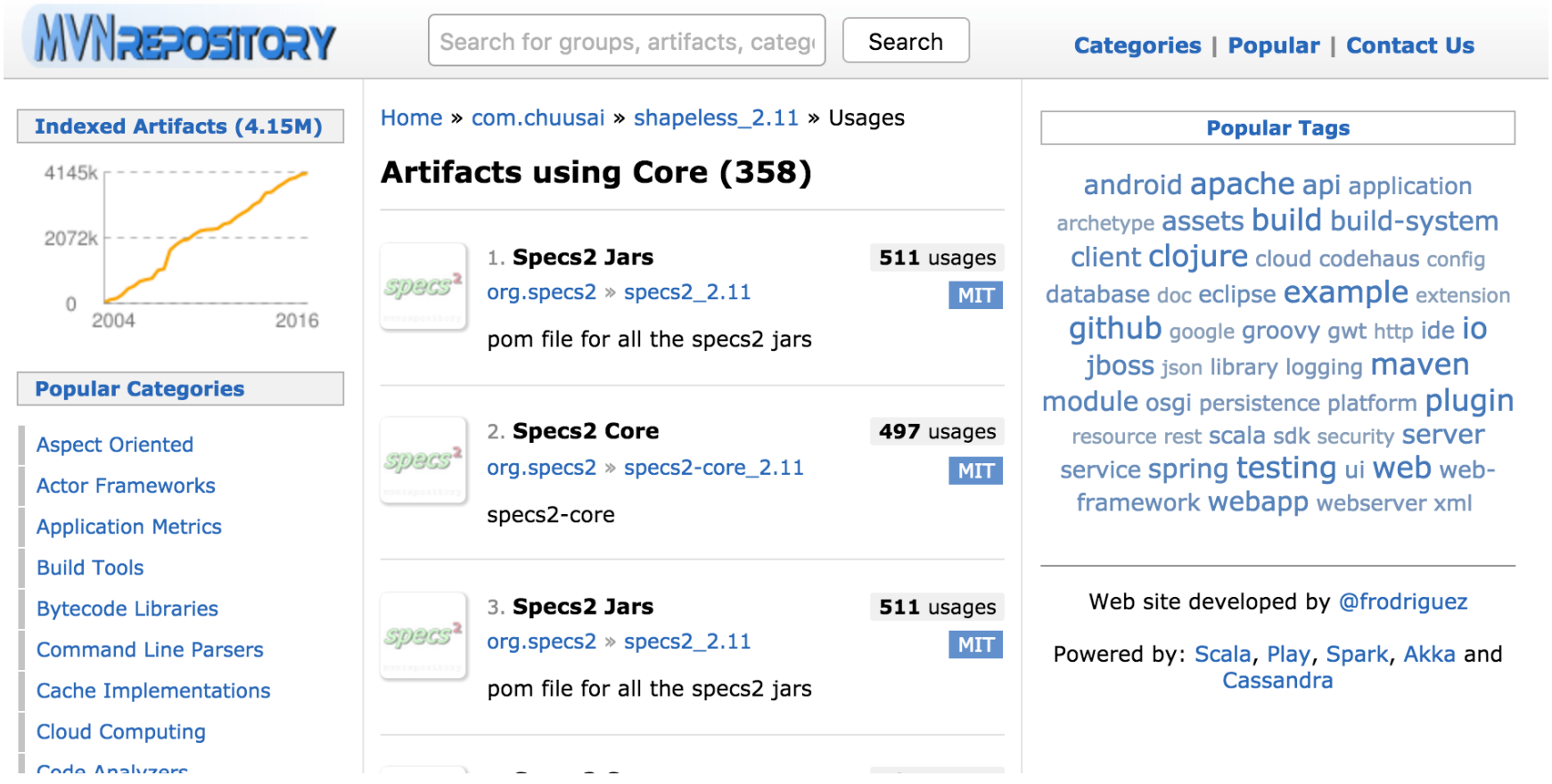
Shapeless 101

twitter: @mermigx (Harry Laou)

Shapeless 101

Slides : http://harrylaou.com/slides/shapeless101.pdf

Libraries that use shapeless



Libraries that use shapeless

- **Specs2** : software specifications - testing
- **Spray Routing** : RESTful web services on top of Akka
- **Breeze** : numerical processing
- **Ammonite** : scala scripting

Libraries that use shapeless

- **Circe** : JSON library that provides generic codec derivation using Shapeless.
- **Ficus** : wrapper companion for Typesafe config
- **ReactiveMongo** : non-blocking and asynchronous mongo driver
- **Finch** : purely functional basic blocks atop of Finagle

Libraries that use shapeless

- **Refined** : refinement types with type-level predicates
- **Parboiled** : parsing library for arbitrary input text based on Parsing expression grammars
- **Monocle** : Optics library
- **Phantom** : driver for Cassandra

Libraries that use shapeless

- **Scalacheck** : automated property-based testing
- **Quasar** : NoSQL analytics engine
- **Enum** : enumeration toolbox for Scala
- **Ensime** : IDE-like features to a text editor

Libraries that use shapeless

- **Scodec** : combinator library for working with binary data
- **Anorm** : data access layer
- **Doobie** : principled database access
- **Scala Exercises** : learning tool

What we will cover

- Heterogenous Lists
- The Generic[T] object
- Polymorphic functions

What we will cover

- Natural Transformations
- Product and Coproduct
- The Aux Pattern

What we will cover

- Witness type
- Singleton type
- LabelledGeneric
- example: how to convert the parameters of a case class to a map using shapeless.

Type-level scala programming

just a hint

```
def bla(v:String):String = ???
// v is value parameter

def bla2[A](v:A):A = ???
// v is value parameter and A is type parameter
```

Type-level scala programming

more at

Type-Level Programming in Scala

Heterogenous Lists

A List containing elements of different types and retains the type of each element in its own type

```
sealed trait HList
case object HNil extends HList
case class ::[H, T <: HList](head: H, tail: T)
            extends HList
```

Heterogenous Lists

How to use it ?

```
import shapeless._

val hlist = 26 :: "Harry" :: HNil
```

Heterogenous Lists

HList methods

- `map`
- `flatMap`
- `foldLeft` (and other fold operations)
- `zipper`
- `unify`
- `toList`

Heterogenous Lists

more at

Shapeless features overview : Heterogenous lists

The Generic[T] object

A `Generic[T]` object implements the methods

- `to(t:T):HList`
- `from(hl:HList):T`

for a given product type (usually, a case class or a tuple).

The Generic[T] object

case class

```
import shapeless._

case class Person(name:String, age:Int)
val gp = Generic[Person]
val harry = Person("Harry", 39)
val hl: String :: Int :: HNil = gp.to(harry)
val p: Person = gp.from(hl)
assert( harry == p )
```

The Generic[T] object

tuple

```
import shapeless._

val gp = Generic[Tuple2[String,Int]]
val harryTuple = ("Harry", 39)
val hl: String :: Int :: HNil = gp.to(harryTuple)
val tp: Tuple2[String,Int] = gp.from(hl)
assert(harryTuple == tp)
```

Higher Kinds and Type Constructors

`List`

is a **type constructor**, it takes one type parameter

notation: `* -> *`

--

`List[String]`

is a **type**, produced by a type constructor `List` using type parameter `String`

notation: `*`

Higher Kinds and Type Constructors

Type Constructors

Option, Future, F[_]

Types

Option[Int], Future[String], F[A]

Natural Transformations

```
trait NatT[F[_], G[_]] { def apply[T](f : F[T]) : G[T] }
```

or

```
F ~> G
```

This is very similar to Function1 but for type constructors.

Natural Transformations

Function1

```
val f : A => B = ???
```

Natural Transformation

```
val nat : F ~> G
```

or

```
val nat : F[A] => G[A]
```

Polymorphic functions

a simple case of natural transformation

```
import poly._

object choose extends (Set ~> Option) {
  def apply[T](s : Set[T]) = s.headOption
}
```

```
scala> choose(Set(1, 2, 3))
res0: Option[Int] = Some(1)

scala> choose(Set('a', 'b', 'c'))
res1: Option[Char] = Some('a')
```

Polymorphic functions

via type classes

```
// size is a function from Ints or Strings or pairs to a
// by type specific cases

object size extends Poly1 {
  implicit def caseInt = at[Int](x => 1)
  implicit def caseString = at[String](_.length)
  implicit def caseTuple[T, U]
    (implicit st : Case.Aux[T, Int],
              su : Case.Aux[U, Int]) =
      at[(T, U)](t => size(t._1)+size(t._2))
}
```

Polymorphic functions

```
scala> size(23)
res4: Int = 1

scala> size("foo")
res5: Int = 3

scala> size((23, "foo"))
res6: Int = 4

scala> size(((23, "foo"), 13))
res7: Int = 5
```

Polymorphic functions

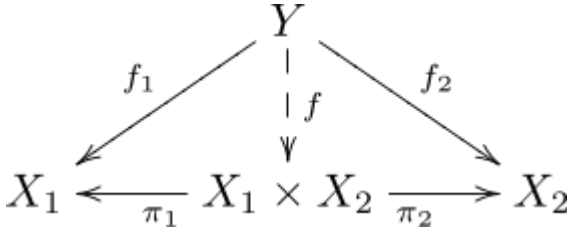
```
//Polymorphic addition with type specific cases
object plus extends Poly2 {
    implicit val caseInt = at[Int,Int](_+_)
    implicit val caseDouble = at[Double,Double](_+_)
    implicit val caseString = at[String,String](_+_)
    implicit def caseList[T] = at[List[T],List[T]](_:::_)
}
```

```
scala> plus(1,1)
res1: plus.caseInt.Result = 2

scala> val p = plus ("foo","bar")
p: plus.caseString.Result = foobar

scala> val p1 :String = plus ("foo","bar")
p1: String = foobar

scala> p1==p
res6: Boolean = true
```

Product and Coproduct

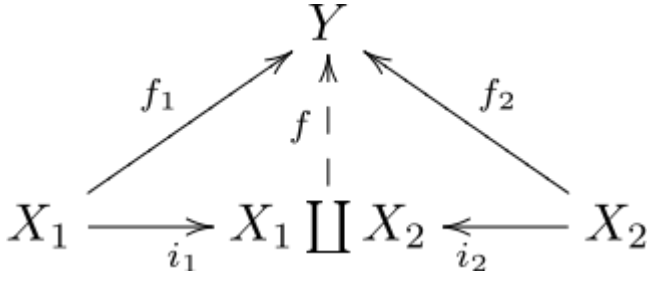
Product

From Wikipedia: In category theory, the product of two (or more) objects in a category is a notion designed to capture the essence behind constructions in other areas of mathematics such as the cartesian product of sets, the direct product of groups, the direct product of rings and the product of topological spaces.



Coproduct

From Wikipedia: In category theory, the coproduct, or categorical sum, is a category-theoretic construction which includes as examples the disjoint union of sets and of topological spaces, the free product of groups, and the direct sum of modules and vector spaces.



Product and Coproduct

What ????

Product

Think of `AND`

example: A tuple is a product

```
val product : (String,Int) = ???
```

Coproduct

Think of `OR`

example : sealed traits

```
sealed trait Animal
case class Dog(name:String) extends Animal
case class Cat(name:String) extends Animal
```

Product and Coproduct

Learn the **different names** of the Product and Coproduct . For example when you need to import `Keys` typeclass for a case class and the ide suggests **shapeless.ops.record._** and **shapeless.ops.union._** in order to choose you need to know

Product and Coproduct

Product	Coproduct
record	sum (type)
tuple	(tagged / disjoint) union

Product and Coproduct

more resources

Alissa Pajer - Products, limits and more!

Category Theory for the Working Hacker

The Aux Pattern

A simple type with one type parameter

```
trait Foo[A] {
  type B
  def value : B
}
```

The Aux Pattern

So let's define some instances:

```
implicit def fi = new Foo[Int] {
  type B = String
  val value = "Foo"
}

implicit def fs = new Foo[String] {
  type B = Boolean
  val value = false
}
```

The Aux Pattern

```
  def foo[T](t: T)(implicit f: Foo[T]): f.B = f.value

  val res1: String = foo(2)
  val res2: Boolean = foo("")
```

So far, so good

The Aux Pattern

But what happens if we want to use the **return type** as a **type parameter** in a type constructor ???

```
import scalaz._, Scalaz._

  def foo[T](t: T)
            (implicit f: Foo[T],
            m: Monoid[f.B]): f.B = m.zero
```

gives compiler error

```
illegal dependent method type: parameter appears in the
type of another parameter in the same section
or an earlier one
```

The Aux Pattern

`Aux` is just a way to extract the result of a type level computation.

```
type Aux[A0, B0] = Foo[A0] { type B = B0  }
```

and it works now:

```
def foo[T, R](t: T)(implicit f: Foo.Aux[T, R],
                            m: Monoid[R]): R = m.zero
  val res1: String = foo(2)

  val res2: Boolean = foo("")
```

The Aux Pattern

more resources

The Aux Pattern (A short introduction to the Aux pattern)

Aux Pattern Evolution

Singleton type

```
val a:Int = 26
```

This is an `Int` . Right ?

Internally in scalac represented as `Int(26)`

But only internally, we cannot access this type.

Singleton type

But we can use shapeless Singleton type

```
import shapeless._, syntax.singleton._

val b = 26.narrow
```

```
b: Int(26) = 26
```

b is of type `Int(26)`

which is a subclass of Int `Int(26) <: Int`

Singleton type

- A singleton type is a type for which there exists exactly one value. ex: `Int(26)`
- we can now lift values into the type system.

Witness

We can go also the other way.

To get a value from a singleton type.

With `Witness`.

Witness

If we have a witness for a singleton type in scope

It will produce the value corresponding to a singleton type.

```
scala> val c = Witness(26)
c: shapeless.Witness.Aux[Int(26)] = shapeless.Witness$$an


scala> c.value
res3: c.T = 26
```

LabelledGeneric

Do you remember `Generic` ?

Labellegeneric is similar but with "labels". :)

So it also implements the methods

- **`to(t:T):HList`**
- **`from(hl:HList):T`**

LabelledGeneric

```
case class Rectangle(width: Int, height: Int)

val genRectangle = LabelledGeneric[Rectangle]

val hlist = genRectangle.to(Rectangle(1,3))
//nothing changed until here

hlist.get('width)
```

```
res1: Int = 1
```

LabelledGeneric

```
val modified = repr.updated('height, 42)

genRectangle.from(modified)
```

```
res4: Rectangle = Rectangle(1,42)
```

LabelledGeneric

another example

```
import shapeless._
import shapeless.ops.record._

case class Foo(bar: String, baz: Boolean)

val labl = LabelledGeneric[Foo]
val keys = Keys[labl.Repr].apply
```

```
scala> println(keys)
'bar :: 'baz :: HNil
```

```
scala> println(keys.toList.map(_.name))
List(bar, baz)
```

LabelledGeneric

more for LabelledGeneric - Singleton Type - Witness Type

labelled generic example in github shapeless

Shapeless : not a tutorial - part 2

Shapeless for Mortals

example

How to convert the parameters of a case class to a map using shapeless.

You could use shapeless.

example

Let

```
case class X(a: Boolean, b: String, c: Int)

case class Y(a: String, b: String)
```

example

Define a LabelledGeneric representation

```
import shapeless._
import shapeless.ops.product._
import shapeless.syntax.std.product._

object X {
  implicit val lgenX = LabelledGeneric[X]
}

object Y {
  implicit val lgenY = LabelledGeneric[Y]
}
```

example

Define a typeclass to provide the toMap methods

```
object ToMapImplicits {
  implicit class ToMapOps2[A <: Product](val a: A)
  extends AnyVal {
    def mkMap(implicit toMap: ToMap.Aux[A, Symbol, Any])
      a.toMap[Symbol, Any]
      .map { case (k: Symbol, v) =>
      k.name -> v.toString }
  }
}
```

example

Then you can use it like this.

```
object Run  extends App {
  import ToMapImplicits._
  val x: X = X(true, "bike",26)
  val y: Y = Y("first", "second")

  val mapX: Map[String, String] = x.mkMap
  val mapY: Map[String, String] = y.mkMap
  println("mapX = " + mapX)
  println("mapY = " + mapY)
}
```

which prints

```
mapX = Map(c -> 26, b -> bike, a -> true)

mapY = Map(b -> second, a -> first)
```

example

This works for simple case classes.

For nested case classes, (thus nested maps) check this answer in SO

Bonus : Scala as Prolog

From A shapeless primer:

how *implicit variables*, *implicit functions*, *type parameters in functions* and *implicit parameter lists of functions* can be "interpreted" in a rule-based context.

Scala	**Prolog**
implicit vals	facts
implicit defs	rules
type parameters in def	variables in a rule
implicit parameter lists	bodies of a rule

Shapeless 101 - Conclusions

- You don't know shapeless (yet)
- You should know the basics and should start making sense
- You can copy some code from Stackoverflow and undestand some of it.

What we saw

- Heterogenous Lists
- The Generic[T] object
- Polymorphic functions

What we saw

- Natural Transformations
- Product and Coproduct
- The Aux Pattern

What we saw

- Witness type
- Singleton type
- LabelledGeneric

Shapeless 101 - Next

- Blog post about introduction resources
- Shapeless guide
- Shapeless gitter channel