Shapeless- Head First

Andreas Koestler - 23.06.2016
Munich Scala User Group

https://github.com/AndreasKostler/scalamuc

Who am I?

"It's an adjustable mortgage. If interest rates go up, your payment increases. If interest rates go down, your payment increases."

What is shapeless?

https://github.com/milessabin/shapeless: “**shapeless** is a type class and dependent type based generic programming library for Scala.”

```
commit 89cbccfd07598bbddc8bf47a4ac23c07690b92f0
Author: Miles Sabin <miles@milessabin.com>
Date:   Tue Sep 13 23:43:49 2011 +0100

    Initial commit.
```

What is shapeless?

- Collection of tools for library authors
  - HList, Coproduct, Poly
- Collection of idioms for type level programming
  - Programming the type system

Type Classes

- Forget everything you know about classes
- Type classes group/categorise types
- Oliveria et. al.:“Type classes were originally developed in Haskell as a disciplined alternative to ad-hoc polymorphism”
- Way of abstracting over types

```
sum(List(1,2,3))

def sum(is: List[Int]) =
  is.foldLeft(0)(_ + _)

def sum(ss: List[String]) =
  ss.foldLeft("")(_ + _)
```

```scala
trait Monoid[A] {
  val zero: A
  def op(a: A, b: A): A
}
```

```scala
def sum[A](as: List[A])(
  implicit M: Monoid[A]
) = as.foldLeft(M.zero)(M.op)
```

```scala
implicit val intMonoid =
  new Monoid[Int] {
    val zero = 0
    def op(a: Int, b: Int) =
      a + b
}

implicit val stringMonoid =
  new Monoid[String] {
    val zero = ""
    def op(a: String, b: String) =
      a + b
}

sum(List(1,2,3)) // => 6
sum(List("a", "b", "c")) // => abd
```

Dependent types

```
trait Foo {
  type Out
  val value: Out
}

def foo(x: Foo): x.Out = x.value
```

Type Members

- Use whenever a type is determined by a type class's type parameters

- If we want to use type members as if they were type parameters we need `Aux`

Dependent types - limitations

```scala
// Cannot be used in computational continuation
def foo[F[_]](f: F[x.Out])(implicit x: Foo) = ???
```

```scala
// Cannot be used in implicits (without using Aux)
def foo(implicit x: Foo, y: Bar[x.Out]) = ???
```

Aux type

```
trait Foo[A] {
  type Out
}
object Foo {
  type Aux[A, Out0] =
    Foo[A] { type Out = Out0 }
  // ...
}
```

Aux type

- Syntactic convenience

```
def foo[A, Out0](implicit f: Foo[A] { type Out = Out0 }) = ???
```

- Don’t need to know the name of the type member
- Less syntactic noise
- Lifts type member back into parameter position

```
def foo[A, Out](implicit f: Foo.Aux[A, Out], b: Bar[Out]) = ???
```

Coproduct

```scala
sealed trait :+:[A, B]

case class Inl[A, B](a: A) extends :+:[A, B]
case class Inr[A, B](b: B) extends :+:[A, B]
```

```scala
// type C = Either[Int, String]
type C = Int :+: String
// val i: C = Left(42)
val i: C = Inl(42)
// val s: C = Right("Foo")
val s: C = Inr("Foo")
```

Does this generalise to more than two types, I wonder?

```scala
sealed trait Coproduct
sealed trait :+:[H, T <: Coproduct] extends Coproduct
sealed trait CNil extends Coproduct

case class Inl[H, T <: Coproduct](head : H) extends :+:[H, T]
case class Inr[H, T <: Coproduct](tail : T) extends :+:[H, T]
```

```scala
type C = Int :+: String :+: Boolean :+: CNil

val i: C = Inl(42)

val s: C = Inr(???)
```

```
// type C = :+:[Int, :+:[String, :+:[Boolean, CNil]]]

type T = String :+: Boolean :+: CNil

type H = Int
```

```
val r: T = Inl("foo")

// Inr(Inl("foo"))
val s: H :+: T = Inr(r)
```

Inject

- Provide a coproduct constructor
- Define Inject type class
    - Implicit proof
- For a given type, prove it can be injected
- As part of the proof, perform the injections

Coproduct `map`

```
trait Functor[F[_]] {
  def map[A, B](as: F[A])(f: A => B): F[B]
}
```

Poly

- Set of functions indexed by Type
  - Function chosen by application type
- Indexing resolved at compile time
  - Compiler error if missing function case
  - Compiler error if applied to non-existent type

```scala
object addOne extends Poly {
  implicit val intCase = at[Int] { a => a + 1 }
  implicit val stringCase = at[String] { s => s + "1" }
}
```

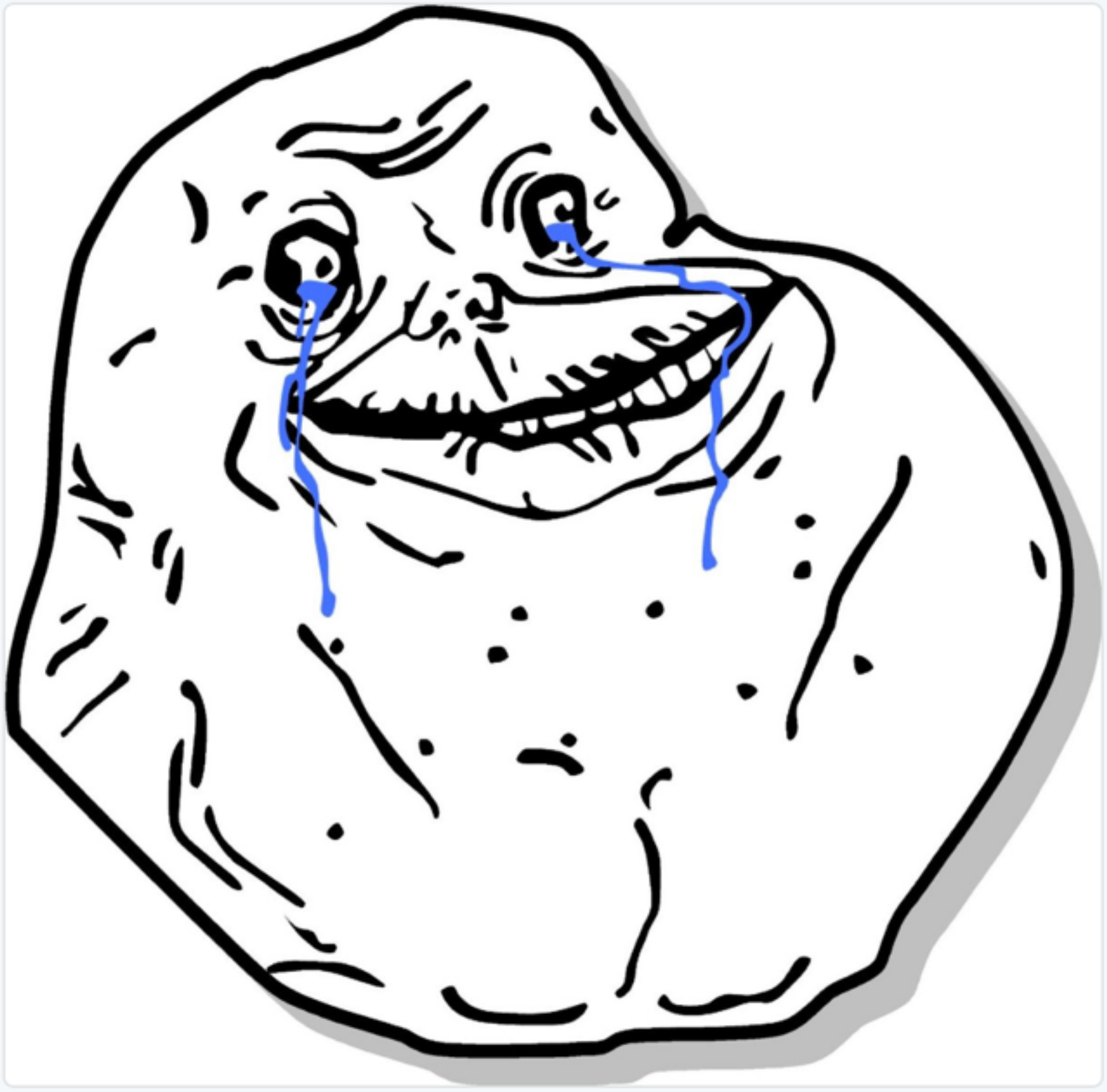
```scala
addOne(42) // 43

addOne("foo") // foo1
```

addOne

```
trait Poly {
  def apply[A, B](a: A)(implicit C: this.Case[A, B]): B = C(a)

  class MkPoly[A] {
    def apply[B](f: A => B): Case[A, B] = new Case[A, B] {
      def apply(a: A) = f(a)
    }
  }

  def at[A] = new MkPoly[A]

  trait Case[A, B] {
    def apply(a: A): B
  }
}
```

Dale Wijnand @dwijnand · Jun 16

In Scala, objects are their own companions...



Downsides

- addOne is an object; objects can't be parameterised

- No anonymous syntax; again, has to be an object

Back to map

- Provide an implicit syntax class with map
- Define Mapper typeclass
  - Implicit proof
- For a given Poly, prove that it can be applied to our coproduct
- Apply the Poly

Paeno numbers

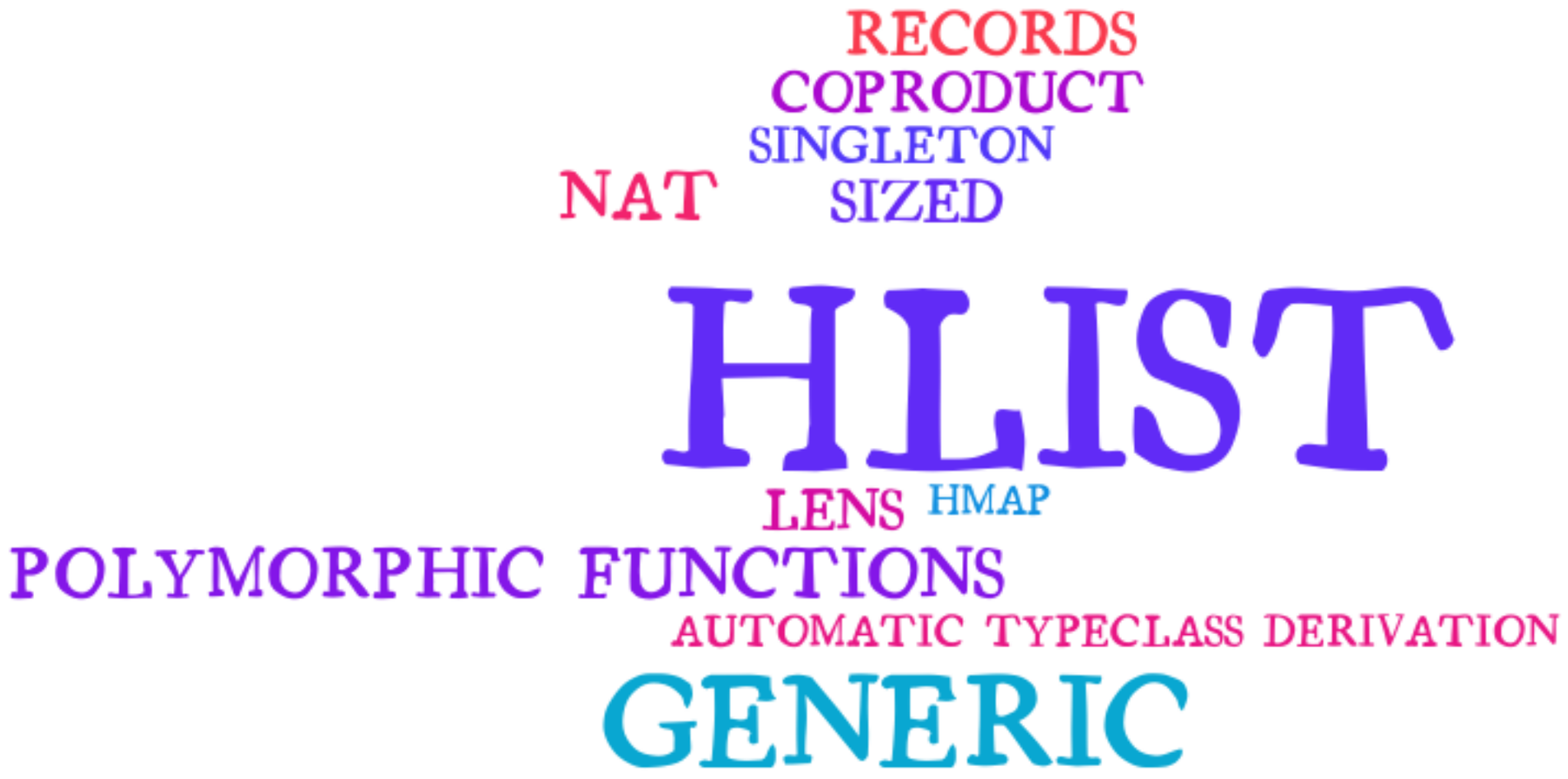
```scala
trait Nat
class _0 extends Nat
class Succ[P <: Nat] extends Nat

type _1 = Succ[_0]
type _2 = Succ[_1]
//...
trait Add[N <: Nat, M <: Nat] {
  type Out <: Nat
}

object Add {
  implicit def baseCase[M <: Nat] = new Add[_0, M] {
    type Out = M
  }

  implicit def recCase[N <: Nat, M <: Nat](implicit a: Add[N, Succ[M]]) =
    new Add[Succ[N], M] {
      type Out = a.Out
    }
}
```

Prolog

- Declarative logic programming language
- Proofs are expressed as facts and rules
- Typelevel programming in Scala is logic programming in scala - George Leontiev
  - Facts: Implicit vals
  - Rules: Implicit defs

```
gcd(X, X, X).
gcd(X, Y, Out) :-
        X < Y,
        Z is Y - X,
        gcd(X, Z, Out).

gcd(X, Y, Out) :-
        Y < X,
        gcd(Y, X, Out).
```

Much, much more



Production Codes

- https://github.com/CommBank/coppersmith
  - library to enable the joining, aggregation, and synthesis of “features
- https://github.com/CommBank/eventually
  - library for event-related analytics, built on top of Scalding
- https://github.com/CommBank/grimlock
  - library for performing data-science and machine learning related data preparation, aggregation, manipulation and querying tasks

Resources

1. Jonathan Merritt: “Discovering Types (from Strings) with Cats and Shapeless” - https://www.youtube.com/watch?v=RecGZB3tcIE

2. Olivieria, Moors, Odersky: “Type Classes as Objects and Implicits” - https://infoscience.epfl.ch/record/150280/files/TypeClasses.pdf

3. Daniel Spiewack: “Roll your own shapeless” - https://vimeo.com/165837504

4. George Leontiev: “There’s a Prolog in your Scala” - https://speakerdeck.com/folone/theres-a-prolog-in-your-scala

Thank you!

@AndreasKoestler

https://github.com/CommBank

https://github.com/AndreasKostler

Type level functions/ Higher kinded types

```
// A => A
def id[A](x: A) = a


// * => *
type Id[A] = A
```

Type level functions

```
// ((A => A) , A) => A
def apply[A](f: A => A, x: A) = f(a)
```

```
// (( * => *) x *) => *
type Apply[F[X], T] = F[T]
```

Eventually

```
type E = A :+: B :+: CNil

def query =
  Select[E]
    .q((One[B], More[A], One[B]) where (
      (_: B, as: Queue[A], b: B) => b.value > average(as.map(_.value))))
    .within(1 minute)
    .compile

val as =
  ThermometerSource(abs) // TypedPipe[E]
    .groupByEntity
    .matchEvents(query)
    .toTypedPipe
    .collect {
    case (_, Some((b, as, c))) => (b.id, as map (_.id), c.id).toString }
```