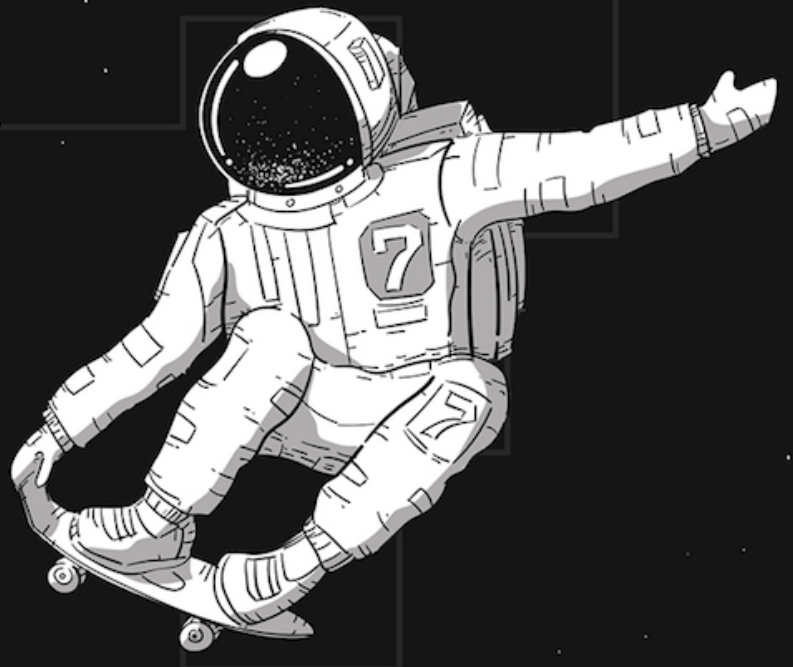


Demystifying Type Class Derivation in Shapeless

Serialize case class?

CSV/JSON/XML/...

```
case class User(id: Long, name: String, active: Boolean)

...

def toCSV[T](t: T) = ?
```

Serialize case class?

CSV/JSON/XML/...

```
case class User(id: Long, name: String, active: Boolean)

...

def toCSV(u: User) = s"${u.id},{u.name},{u.active}"
```

Result:

```
scala> toCSV(User(1L, "test", true))

res1: String = 1,test,true
```

Serialize case class?

CSV/JSON/XML/Whatever

```
case class User(id: Long, name: String, role: Role, addr: Address, score: Double)
sealed trait Role
case class Address(street: String, house: Int)

...

def toCSV(u: User) = s"${u.id}, ${u.name}, ${toCSV(u.role)}, ${toCSV(u.addr)}, ${u
def toCSV(r: Role) = r match { case a: Admin => s"${a.id}, "; case c; Client => s"
def toCSV(r: Address) = s"${a.street}, ${a.house}"
```

Result:

```
scala> toCSV(User(1L, "test", Admin(1L), Address("Wall str", 192), 1.0))
res1: String = 1, test, 1, Wall str, 192, 1.0
```

Problem

```
def toCSV[T](t: T): String = ???
```

Agenda

- Basics: ADTs, Products & Coproducts
- Type Class pattern
- Generic Derivation
- `[live-code]`
- Lazy, Aux
- Debugging

How do we model our domain?

How do we model our domain?

case class / sealed trait

- **case class**
- **sealed trait**

```
case class User(id: Long, name: String,
                active: Boolean, role: Role)

sealed trait Role

case class Admin(id: Long, special: Boolean) extends Role
case class Client(id: Long) extends Role
```

How do we model our domain?

case class / sealed trait

- **TupleN[...] extends Product**
- **scala.Either / cats.Xor / scalaz.\/**

TupleN / Either

```
type User = (Long, String, Boolean, Role)


type Role = Either[Admin, Client]
type Admin = (Long, Boolean)
type Client = Long
```

How do we model our domain?

case class / sealed trait

- **TupleN[...] extends Product**
- **scala.Either / cats.Xor / scalaz.\/**

TupleN / Either

```
type User = (Long, String, Boolean, Role)


type Role = Either[Admin, Client]
type Admin = (Long, Boolean)
type Client = Long
```

Abstracting over arity

case class / sealed trait

TupleN / Either

HList / Coproduct

- **HList**
- **Coproduct**

```
type User = Long :: String :: Boolean :: Role :: HNil

type Role = Admin :+: Client :+: CNil

type Admin = Long :: Boolean :: HNil
type Client = Long :: HNil
```

```
val admin: Role = Inl(2L :: true :: HNil)
val sandy: User = 1 :: "Sandy" :: true :: admin :: HNil
// res5: User = 1 :: Sandy :: true :: Inl(2 :: true :: HNil)
```

ADT *

AND types	**OR** types
case class	**sealed trait**
Tuple	**Either**
HList	**Coproduct**

```
case class User(id: Long, name: String, active: Boolean, role: Role)

sealed trait Role
case class Admin(id: Long, special: Boolean) extends Role
case class Client(id: Long) extends Role
```

* Stands for Algebraic Data Type

HList

```
val hlist = 1 :: "one" :: 1L :: HNil
// res0: shapeless.::[
//          Int,shapeless.::[
//              String,shapeless.::[Long,
//                  shapeless.HNil
//              ]
//          ]
//      ] = 1 :: "one" :: 1L :: HNil

hlist.head // Int
hlist.tail.head // String
hlist.tail.tail.head // Long

val s = hlist.select[String] // returns "one".
demo.select[List[Int]] // Compilation error. demo does not contain a List[Int]
```

```
take, head, tail, map, flatMap, zip
```

HList

Definition:

```
sealed trait HList

case class ::[H, T <: HList](head : H, tail : T) extends HList // HCons

case object HNil extends HList
```

HList

Definition:

```
sealed trait HList

case class ::[H, T <: HList](head : H, tail : T) extends HList // HCons

case object HNil extends HList
```

List Definition:

```
sealed trait List[+A]

case class ::[A](head: A, tail: List[A]) extends List[A]

case object Nil extends List[Nothing]
```

Coproduct

Once you feel comfortable with an HList - for Coproduct it is quite similar

```
sealed trait Coproduct

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

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

sealed trait CNil extends Coproduct
```

Usage:

```
// 'kind-of' Either[Int, String, Boolean]
type Co = Int :+: String :+: Boolean :+: CNil

val co1 = Inl(42L) // Int :+: String :+: Boolean :+: CNil
val co2 = Inr(Inl("forty-two") // Int :+: String :+: Boolean :+: CNil
val co3 = Inr(Inr(Inl(true)) // Int :+: String :+: Boolean :+: CNil
```

shapeless.Generic

Generic

```
import shapeless.Generic

case class User(id: Long, name: String, active: Boolean)

val generic = Generic[User]

// res0: shapeless.Generic[User]{type Repr = shapeless.::[Long,
//  shapeless.::[String,
//      shapeless.::[Boolean,
//          shapeless.HNil
//      ]
//  ]
//]}
```

Generic

```
trait Generic[A] {

  type Repr

  def to(value: A): Repr

  def from(value: Repr): A
}
```

Usage:

```
scala> val user = User(1L, "josh", true)
user: User = User(1,josh,true)

scala> generic.to(user)
res2: res1.Repr = 1 :: josh :: true :: HNil

scala> generic.from(res2)
res3: User = User(1,josh,true)
```

Generic

```
trait Generic[A] {

  type Repr

  def to(value: A): Repr

  def from(value: Repr): A
}
```

Usage:

```
scala> val user = User(1L, "josh", true)
user: User = User(1,josh,true)

scala> generic.to(user)
res2: res1.Repr = 1 :: josh :: true :: HNil

scala> generic.from(res2)
res3: User = User(1,josh,true)
```


Type Class
pattern

Type Class pattern

scala standard library

- Numeric (Collection's `sum, product, min, max`)

- Ordering (Collection's `sorted`)

- CanBuildFrom (whole collections api)

- IsSeqLike

- IsTraversableOnce

Cats, Scalaz (all of the functional abstractions like Functor, Applicative, Monad, ...)

Type Class pattern

scala.Ordering

```scala
/**
  * Ordering is a trait whose instances each represent
  * a strategy for sorting instances of a type.
  * ...
  */

trait Ordering[T] extends Comparator[T] {

 /**
   * Returns an integer whose sign communicates
   * how x compares to y.
   */
  def compare(x: T, y: T): Int
}
```

Type Class pattern

scala.Ordering

```
trait Ordering[T] extends Comparator[T] {
  def compare(x: T, y: T): Int
}
```

Definition

Type Class pattern

scala.Ordering

Definition

```
trait Ordering[T] extends Comparator[T] {
  def compare(x: T, y: T): Int
}
```

Instances

```
object Ordering {
  implicit val intOrd: Ordering[Int] = new Ordering[Int] {
    def compare(x: Int, y: Int) = lang.Integer.compare(x, y
  }
  implicit val longOrd: Ordering[Long] = new Ordering[Long]
    def compare(x: Long, y: Long) = lang.Long.compare(x, y)
  }
  ...
}
```

Type Class pattern

scala.Ordering

Definition

Instances

implicit parameter

```
trait Ordering[T] extends Comparator[T] {
  def compare(x: T, y: T): Int
}
```

```
object Ordering {
  implicit val intOrd: Ordering[Int] = new Ordering[Int] {
    def compare(x: Int, y: Int) = lang.Integer.compare(x, y
  }
  implicit val longOrd: Ordering[Long] = new Ordering[Long]
    def compare(x: Long, y: Long) = lang.Long.compare(x, y)
  }
  ...
}
```

```
def sorted[T](implicit ord: Ordering[T]): Repr
```

..or context bound

```
def sorted[T: Ordering]: Repr
```

Type Class pattern

scala.Ordering

Definition

Instances

implicit
parameter

Usage:

```
List(1, 3, 2).sorted
// at this point compiler is searching implicit value
// in the ?global?, local scope, imported scope, companion
```

Type Class pattern

https://github.com/mpilquist/simulacrum

```
@typeclass trait CSVSerializer[A] {
  @op("§") def serialize(a: A): String
}
```

Type Class pattern

https://github.com/mpilquist/simulacrum

```
@typeclass trait CSVSerializer[A] {
  @op("§") def serialize(a: A): String
}
```

```
trait CSVSerializer[A] {
  def serialize(x: A, y: A): A
}

object CSVSerializer {
  def apply[A](implicit instance: CSVSerializer[A]): CSVSerializer[A] = instance

  trait Ops[A] {
    def typeClassInstance: CSVSerializer[A]
    def self: A
    def §(y: A): A = typeClassInstance.append(self, y)
  }
  trait ToCSVSerializerOps {
    implicit def toCSVSerializerOps[A](target: A)(implicit tc: CSVSerializer[A]):
      val self = target
      val typeClassInstance = tc
    }
  }
  ...
```

CSV Serializer

Type Class definition

```
trait CSVSerializer[A] {

  def serialize(a: A): List[String]
}
```

CSV Serializer

Type Class definition

```
trait CSVSerializer[A] {

  def serialize(a: A): List[String]
}
```

lets define some helpers..

```
object CSVSerializer {

  def apply[A](implicit serializer: CSVSerializer[A]): CSVSerializer[A] =
                              serializer

  implicit class WithSerialize[A](a: A) {
    def toCSV(implicit serializer: CSVSerializer[A]) =
                          serializer.serialize(a).mkString(",")
  }
}
```

deriving serializer

[live-code]

Aux Pattern

```
scala> trait Foo { type T }
// defined trait Foo

scala> def f(foo: Foo, t: foo.T) = ???

  <console>:13: error: illegal dependent method type: parameter may only be
                                  referenced in a subsequent parameter section
         def f(foo: Foo, t: foo.T) = ???
                ^

scala>
```

Aux Pattern

```
scala> trait Foo { type T }
// defined trait Foo

scala> def f(foo: Foo, t: foo.T) = ???

  <console>:13: error: illegal dependent method type: parameter may only be
                                  referenced in a subsequent parameter section
         def f(foo: Foo, t: foo.T) = ???
                ^
scala>
```

Solution:

```
scala> trait Foo { type T }
// defined trait Foo

scala> type Aux[T0] = Foo { type T = T0 }
// defined type alias Aux

scala> def f[T](foo: Aux[T], t: T) = ???
// f: [T](foo: Aux[T], t: T)Nothing
```

Aux Pattern

```
type Aux[T, Repr0] = Generic[T] { type Repr = Repr0 }
```

Implicit divergence

nested structure

```
case class Account(id: Long, user: User)
case class User(id: Long, name: String, active: Boolean)
```

```
  /*1*/ CSVSerializer[Account]
  /*2*/ CSVSerializer[::[Long, ::[User, HNil]]]
  /*3*/ CSVSerializer[::[User, HNil]]
  /*4*/ CSVSerializer[User]
  /*5*/ CSVSerializer[::[Long, ::[String, ::[Boolean, HNil]]]] // failed

// diverging implicit expansion for type xyz.codefastdieyoung
//        .CSVSerializer[Long :: String :: Boolean :: shapeless.HNil]
```

the compiler sees the same type constructor twice and the complexity of the type parameters is *increasing*...

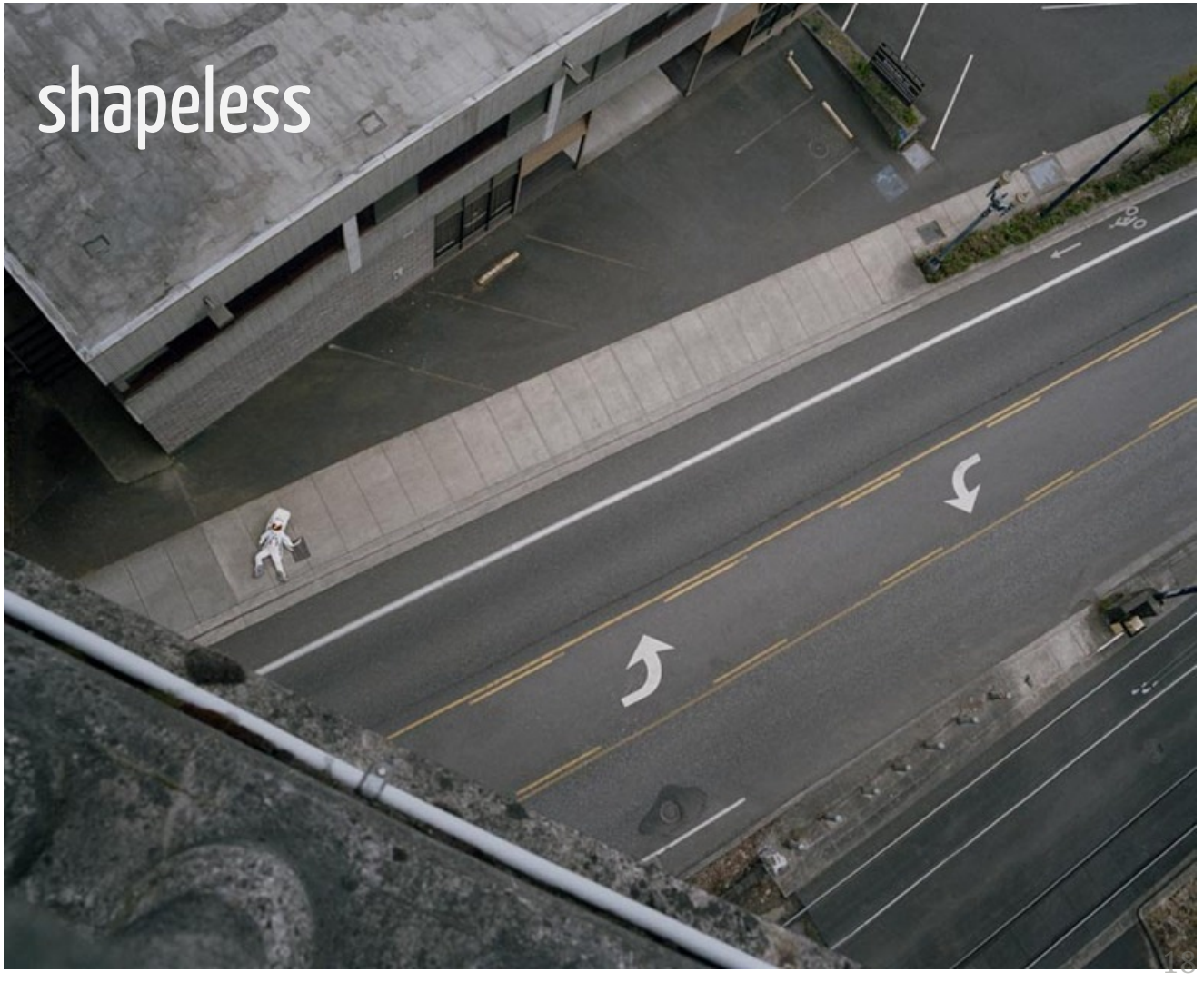
Lazy!

```
import shapeless.Lazy

implicit def HConsCSVSerializer[H, T <: HList](implicit
  hSerializer: Lazy[CSVSerializer[H]],
  tailSerializer: CSVSerializer[T]
): CSVSerializer[H :: T] = { t =>
  s"${hSerializer(t.head)}, ${tailSerializer(t.tail)}"
}
```

wrap diverging implicit parameter...

- it suppresses implicit divergence at compile time
- it defers evaluation of the implicit parameter at runtime


shapeless

Generic programming

```
def fmap[A](f: A => B): F[B] = ???
```


Debugging
implicits

Debugging implicits

- implicitly
- the
- reify
- @showInfix
- and of course - IDE short-cuts to lookup implicit values


MY NOSE ITCHES

Issues

- Compilation Times
- inductive implicits
  - typelevel fork 2.11+
  - `-Yinduction-heuristics`
- runtime overhead
- ...

Conclusion

Poly

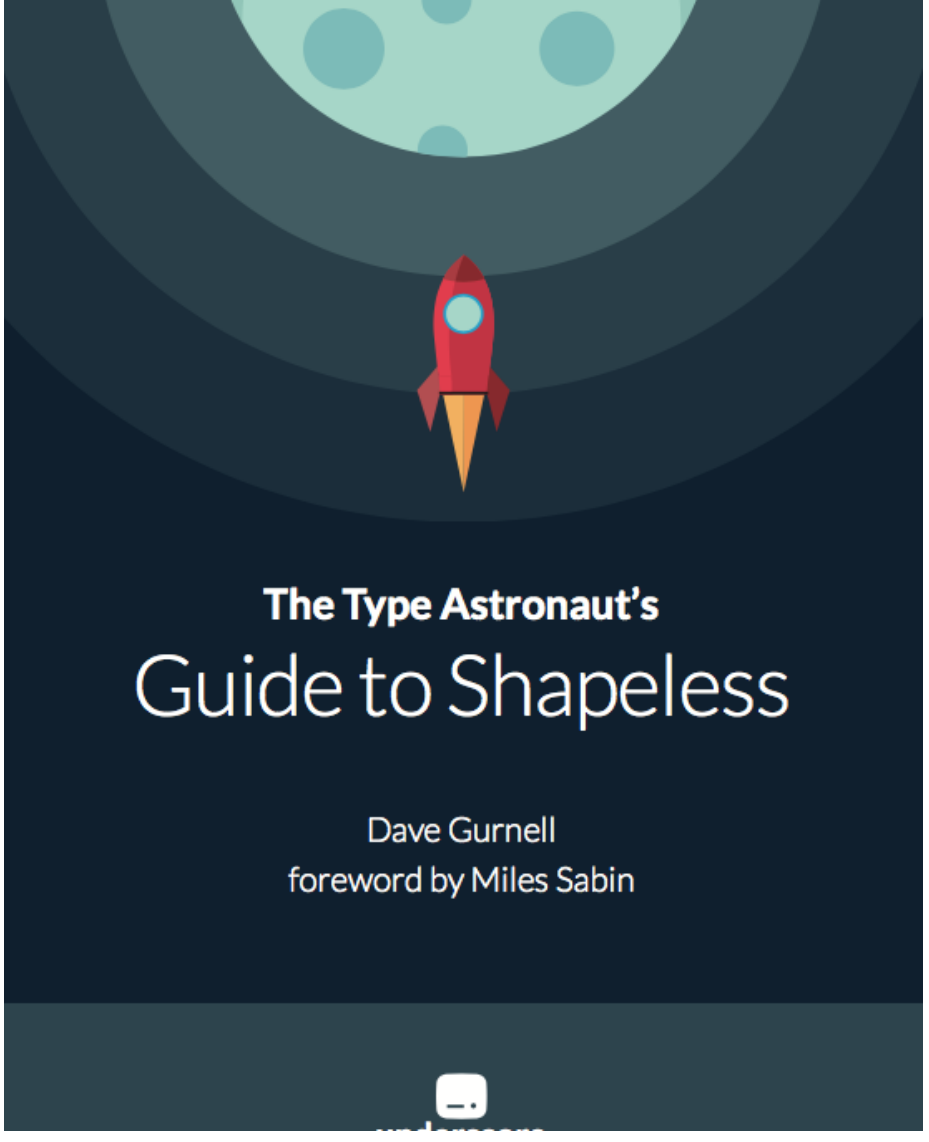
ops

Lenses

TypeClass type class

Peano numbers

Utils (the, not-compile, Typable)


The Type Astronaut's
Guide to Shapeless
Dave Gurnell
foreword by Miles Sabin

Thanks

- @twist522
- github.com/thatwist
- crossroadlabs.xyz
- vote.scalaua.com