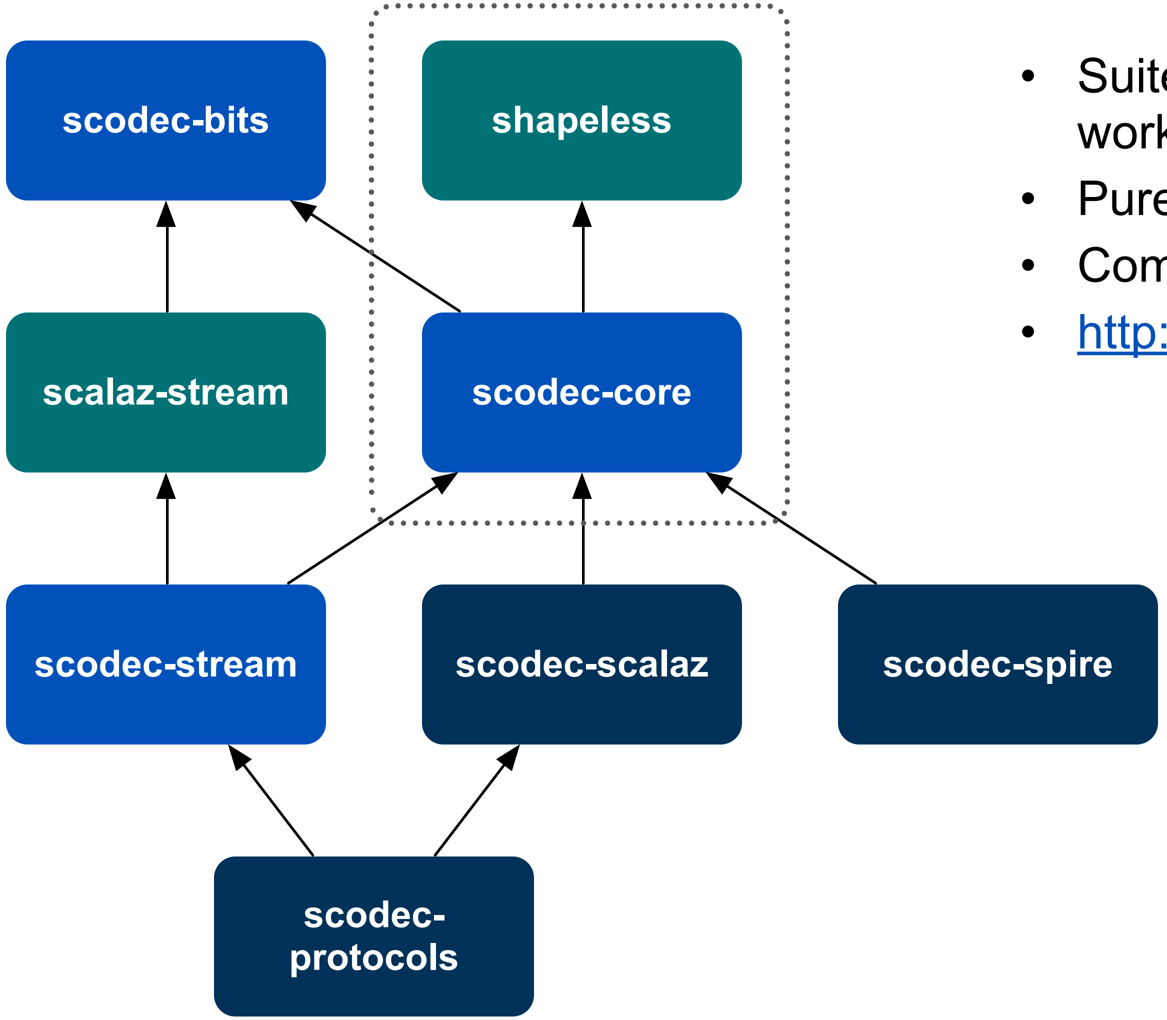
Introduction to generic programming with Shapeless, with applications from scodec

April 2015

About Me…

- Michael Pilquist (@mpilquist)
- Using Scala professionally since 2008
- Author of scodec (personal project)
- Work at CCAD, LLC. (Combined Conditional Access Development)
  - Joint-venture between Comcast and ARRIS Group, Inc.
  - Build conditional access technology

scodec



- Suite of Scala libraries for working with binary data
- Pure functional
- Compositional
- http://scodec.org

Introduction to scodec

```scala
case class Point(x: Int, y: Int)
case class Line(start: Point, end: Point)
case class Arrangement(lines: Vector[Line])

val arr = Arrangement(Vector(
  Line(Point(0, 0), Point(10, 10)),
  Line(Point(0, 10), Point(10, 0))))
```

Introduction to scodec

```
import scodec.Codec
import scodec.codecs.implicits._

val arrBinary = Codec.encode(arr).require

// arrBinary: BitVector = BitVector(288 bits,
0x000000020000000000000000000000000a0000000a00000000000…

val decoded = Codec[Arrangement].
  decode(arrBinary).require.valid

// decoded: Arrangement = Arrangement(Vector(
  Line(Point(0, 0),Point(10, 10)),
  Line(Point(0, 10),Point(10, 0))))
```

What happened here?

- At compile time…
  - Codec[Arrangement] generated, which encoded the length of the line vector in a 32-bit signed big endian field followed by an encoded form of each Line using an implicit Codec[Line]
  - Codec[Line] generated, which encoded the start and end points using an implicit Codec[Point]
  - Codec[Point] generated, which encoded the x and y fields using an implicit Codec[Int]
  - scodec.codecs.implicits._ import provided a Codec[Int] which encoded integers as 32-bit signed big endian fields
- No reflection!

Lots of combinators…

```
import scodec.codecs.zlib

val compressed = zlib(Codec[Arrangement])

val arrBinary = compressed.encode(arr).require

// arrBinary: BitVector = BitVector(152 bits,
0x789c636060606240002e2846610300026e002b)

val decoded = compressed.decode(arrBinary).require.valid

// decoded: Arrangement = Arrangement(Vector(
  Line(Point(0, 0),Point(10, 10)),
  Line(Point(0, 10),Point(10, 0))))
```

Controlling codec derivation with implicits

```
case class Point(x: Int, y: Int)
case class Line(start: Point, end: Point)
case class Arrangement(lines: Vector[Line])

val arr = Arrangement(Vector(
  Line(Point(0, 0), Point(10, 10)),
  Line(Point(0, 10), Point(10, 0))))

import scodec.Codec
import scodec.codecs.implicits.{ implicitIntCodec => _, _ }
implicit val ci = scodec.codecs.uint8

val arrBinary = Codec.encode(arr).require
// arrBinary: BitVector = BitVector(72 bits,
0x0200000a0a000a0a00)
```

Controlling codec derivation with implicits

```
val result = Codec.encode(
  Arrangement(Vector(
    Line(Point(0, 0), Point(1, 1)),
    Line(Point(0, 0), Point(1, -1))
  ))
)

// result: Attempt[BitVector] = Failure(lines/1/end/y: -1
 is less than minimum value 0 for 8-bit unsigned integer)
```

Explicit codecs

```scala
case class TransportStreamHeader(
  transportErrorIndicator: Boolean,
  payloadUnitStartIndicator: Boolean,
  transportPriority: Boolean,
  pid: Int,
  scramblingControl: Int,
  adaptationFieldControl: Int,
  continuityCounter: Int
)
```

Explicit codecs

```
object TransportStreamHeader {
  implicit val codec: Codec[TransportStreamHeader] =
    "transport_stream_header" | fixedSizeBytes(4,
      ("syncByte"                  | constant(0x47)     ) :~>:
      ("transportErrorIndicator"   | bool               ) ::
      ("payloadUnitStartIndicator" | bool               ) ::
      ("transportPriority"         | bool               ) ::
      ("pid"                       | uint(13)           ) ::
      ("scramblingControl"         | uint2              ) ::
      ("adaptationFieldControl"    | uint2              ) ::
      ("continuityCounter"         | uint4              )
    ).as[TransportStreamHeader]
}
```

Shapeless Usage in scodec

- All of the previous examples made *heavy* use of Shapeless
  - HLists
  - Singleton types
  - Records
  - Proofs
  - Automatic type class derivations
- Shapeless was not in the *surface API* in some cases

Shapeless

Skeleton of a List

```
sealed trait List[+A] {
  def ::[AA >: A](h: AA): List[AA] = new ::(h, this)
}

case object Nil extends List[Nothing]

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

Skeleton of a Heterogen(e)ous List

```
sealed trait HList

sealed trait HNil extends HList {

}
case object HNil extends HNil

case class ::[+H, +T <: HList](head: H, tail: T)
  extends HList {

}
```

Skeleton of a Heterogen(e)ous List

```
sealed trait HList

sealed trait HNil extends HList {
  def ::[H](h: H): H :: HNil = new ::(h, this)
}
case object HNil extends HNil

case class ::[+H, +T <: HList](head: H, tail: T)
  extends HList {
    def ::[G](g: G): G :: H :: T = new ::(g, this)
}
```

- Size encoded in the type of the HList
- Type of each element encoded in the type of the HList

List vs HList Construction

```
val xs = 1 :: 2 :: 3 :: Nil
// xs: List[Int] = List(1, 2, 3)
```

```
val xs = 1 :: 2 :: 3 :: HNil
// xs: shapeless.::[Int,shapeless.::[Int,shapeless.::
[Int,shapeless.HNil]]] = 1 :: 2 :: 3 :: HNil
```

Type Operators

Scala allows any binary type constructor to be used with infix syntax

`Map[Int, String]`	`Int Map String`
`Or[Int, String]`	`Int Or String`
`\/[Int, String]`	`Int \/ String`
`::[Int, ::[String, HNil]]`	`::[Int, String :: HNil]`
	`Int :: String :: HNil`

Type Operators

```
val xs = 1 :: 2 :: 3 :: HNil
// xs: shapeless.::[Int,shapeless.::[Int,shapeless.::
[Int,shapeless.HNil]]] = 1 :: 2 :: 3 :: HNil

// xs: shapeless.::[Int,shapeless.::[Int,Int :: HNil]] =
1 :: 2 :: 3 :: HNil

// xs: shapeless.::[Int,Int :: Int :: HNil] = 1 :: 2 :: 3
:: HNil

// xs: Int :: Int :: Int :: HNil = 1 :: 2 :: 3 :: HNil
```

- Transformation is purely mechanical!
- Potential improvement for Scala and/or Typelevel Scala: https://github.com/typelevel/scala/issues/43

List vs HList Construction

```
val xs = 1 :: "hello" :: 3 :: Nil
// xs: List[Any] = List(1, hello, 3)
```

```
val xs = 1 :: "hello" :: 3 :: HNil

// xs: shapeless.::[Int,shapeless.::[String,shapeless.::
[Int,shapeless.HNil]]] = 1 :: hello :: 3 :: HNil

// xs: Int :: String :: Int :: HNil =
  1 :: hello :: 3 :: HNil
```

HList head/tail

```
val xs = 1 :: "hello" :: 3 :: HNil
// xs: shapeless.::[Int,shapeless.::[String,shapeless.::
[Int,shapeless.HNil]]] = 1 :: hello :: 3 :: HNil

xs.head
// res0: Int = 1

xs.tail
// res1: shapeless.::[String,shapeless.::
[Int,shapeless.HNil]] = hello :: 3 :: HNil

xs.tail.head
// res2: String = hello

xs.tail.tail.head
// res3: Int = 3
```

HList head/tail

```
val xs = 1 :: "hello" :: 3 :: HNil
// xs: shapeless.::[Int,shapeless.::[String,shapeless.::
[Int,shapeless.HNil]]] = 1 :: hello :: 3 :: HNil

xs.tail.tail.tail.head
// <console>:12: error: could not find implicit value for
parameter c: shapeless.ops.hlist.IsHCons[shapeless.HNil]
//               xs.tail.tail.tail.head
//                                 ^

scala> :t xs.tail.tail.tail
shapeless.HNil
```

HList Operations: map

```
import shapeless.{ ::, HNil, Poly1 }

val xs = 1 :: "hello" :: 3 :: HNil

object inc extends Poly1 {
  implicit def caseInt = at[Int](_ + 1)
  implicit def default[A] = at[A](a => a)
}

val ys = xs map inc
// ys: shapeless.::[Int,shapeless.::[String,shapeless.::
[Int,shapeless.HNil]]] = 2 :: hello :: 4 :: HNil
```

HList Operations: map

```
import shapeless.{ ::, HNil, Poly1 }

val xs = 1 :: "hello" :: 3 :: HNil

object increv extends Poly1 {
  implicit def caseInt = at[Int](_ + 1)
  implicit def caseString = at[String](_.reverse)
}

val ys = xs map increv
// ys: shapeless.::[Int,shapeless.::[String,shapeless.::
[Int,shapeless.HNil]]] = 2 :: olleh :: 4 :: HNil
```

HList Operations: map

```scala
import shapeless.{ ::, HNil, Poly1 }

val xs = 1 :: "hello" :: 3 :: HNil

object inc extends Poly1 {
  implicit def caseInt = at[Int](_ + 1)
}

val ys = xs map inc
// <console>:10: error: could not find implicit value for
parameter mapper:
shapeless.ops.hlist.Mapper[inc.type,shapeless.::
[Int,shapeless.::[String,shapeless.::
[Int,shapeless.HNil]]]]
       val ys = xs map inc
                   ^
```

HList Operations: take/drop

```
import shapeless.{ ::, HNil }

val xs = 1 :: "hello" :: 3 :: HNil

val ys = xs take 2
// ys: shapeless.::[Int,shapeless.::
[String,shapeless.HNil]] = 1 :: hello :: HNil

val zs = xs drop 2
// zs: shapeless.::[Int,shapeless.HNil] = 3 :: HNil
```

HList Operations: take/drop

```
import shapeless.{ ::, HNil }

val xs = 1 :: "hello" :: 3 :: HNil

val ys = xs take 4
// <console>:10: error: Implicit not found:
shapeless.Ops.Take[shapeless.::[Int,shapeless.::
[String,shapeless.::[Int,shapeless.HNil]]], nat_$macro
$3.N]. You requested to take nat_$macro$3.N elements, but
the HList shapeless.::[Int,shapeless.::
[String,shapeless.::[Int,shapeless.HNil]]] is too short.
              xs take 4
                 ^
```

HList Operations: unify

```scala
sealed trait Parent extends Product with Serializable
case class Foo(value: Int) extends Parent
case class Bar(value: Double) extends Parent
case object Baz extends Parent

val xs = Foo(1) :: Bar(2.0) :: Baz :: HNil
// xs: shapeless.::[Foo,shapeless.::[Bar,shapeless.::
[Baz.type,shapeless.HNil]]] = Foo(1) :: Bar(2.0) :: Baz ::
HNil

val ys = xs.toList
// ys: List[Parent] = List(Foo(1), Bar(2.0), Baz)

val zs = xs.unify
// zs : shapeless.::[Parent,shapeless.::
[Parent,shapeless.::[Parent,shapeless.HNil]]] = Foo(1) ::
Bar(2.0) :: Baz :: HNil
```

HList Operations: unify - implementation

Extension method

```
implicit class HListOps[L <: HList](val l: L) {
  def unify(implicit u: Unifier[L]): u.Out = u(l)
}
```

Operation

Path dependent type

```
sealed trait Unifier[L <: HList] {
  type Out <: HList
  def apply(l: L): Out
}

object Unifier {
  // TODO: Proof by induction on structure of HList
}
```

Simplified version from https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala

HList Operations: unify - proof - base cases

Allows the output type to be bound to a type var without resorting to refinement types

```
object Unifier {

  type Aux[L0 <: HList, Out0 <: HList] =
    Unifier[L0] { type Out = Out0 }

  implicit def forHNil: Unifier.Aux[HNil, HNil] =
    new Unifier[HNil] {
      type Out = HNil
      def apply(l: HNil) = l
    }

  implicit def forOne[H]: Unifier.Aux[H :: HNil, H :: HNil] =
    new Unifier[H :: HNil] {
      type Out = H :: HNil
      def apply(l: H :: HNil) = l
    }
}
```

Simplified version from https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala

HList Operations: unify - proof - inductive case

Type operator that witnesses that HLub is the least upper bound of types H1 and H2

Recursive step

```
object Unifier {

  implicit def forHList[H1, H2, HLub, T <: HList](implicit
    lub: Lub[H1, H2, HLub],
    tailUnifier: Unifier[HLub :: T]
  ): Unifier.Aux[H1 :: H2 :: T, HLub :: tailUnifier.Out] =
    new Unifier[H1 :: H2 :: T] {
      type Out = HLub :: tailUnifier.Out
      def apply(l: H1 :: H2 :: T) =
        lub.left(l.head) ::
          tailUnifier(lub.right(l.tail.head) ::
          l.tail.tail)
    }
}
```

Simplified version from https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala

Type Operator Summary

- A sealed trait that defines:
  - input and output types
  - method(s) that perform the operation(s) in terms of input and output types
- A companion that defines:
  - an Aux type alias that lifts each abstract type member to a type parameter
  - implicit instances of the trait in terms of:
    - base cases
    - inductive cases
  - which often use other type operators

Scalac as a proof assistant

- Scalac is not intended as a proof assistant
- Such proofs must be written to cooperate with such things as:
  - left-to-right implicit parameter resolution
  - lack of backtracking in type parameter inference
  - erroneous implicit divergence
  - unpredictable compilation performance
- For example, swapping the order of the implicit params in the previous example breaks the proof:

```
implicit def forHList[H1, H2, HLub, T <: HList](implicit
  tailUnifier: Unifier[HLub :: T],
  lub: Lub[H1, H2, HLub]
): Unifier.Aux[H1 :: H2 :: T, HLub :: tailUnifier.Out] =
```

despite y'alls best efforts, Scala is not a proof assistant -- sorry.

it is a proof assistant ... just not a very good one

Abstracting over HLists

```
import shapeless._

/**
 * Replaces each number in the input HList with its
 * square, leaving all non-numbers alone.
 */
def square(l: HList): HList = ???
```

Abstracting over HLists

```scala
import shapeless._

def square(l: HList): HList = ???
```

We can’t do anything with just an HList — it is a sealed trait with no members — and the caller can’t do anything the the returned HList for the same reason

Abstracting over HLists

```scala
import shapeless._

def square[L <: HList](l: L): L = ???
```

Introduce a type parameter that captures the shape of the input list

Returned list has the same shape — size and component types — as the input list

Abstracting over HLists

Define a polymorphic function that squares various numeric types and ignores non-numeric types

```
import shapeless._

object sq extends Poly1 {
  implicit def caseInt = at[Int](x => x * x)
  implicit def caseLong = at[Long](x => x * x)
  implicit def caseFloat = at[Float](x => x * x)
  implicit def caseDouble = at[Double](x => x * x)
  implicit def default[A] = at[A](x => x)
}

def square[L <: HList](l: L): L = l map sq
```

Map the polymorphic function over the input HList

Abstracting over HLists

```
import shapeless._

object sq extends Poly1 {
  implicit def caseInt = at[Int](x => x * x)
  implicit def caseLong = at[Long](x => x * x)
  implicit def caseFloat = at[Float](x => x * x)
  implicit def caseDouble = at[Double](x => x * x)
  implicit def default[A] = at[A](x => x)
}

def square[L <: HList](l: L): L = l map sq
```

```
could not find implicit value for parameter mapper:
shapeless.ops.hlist.Mapper[sq.type,L]
```

Abstracting over HLists

```
import shapeless._

object sq extends Poly1 {
  implicit def caseInt = at[Int](x => x * x)
  implicit def caseLong = at[Long](x => x * x)
  implicit def caseFloat = at[Float](x => x * x)
  implicit def caseDouble = at[Double](x => x * x)
  implicit def default[A] = at[A](x => x)
}

def square[L <: HList](implicit
  m: ops.hlist.Mapper[sq.type, L]
)(l: L): L = l map sq
```

Add missing implicit and recompile

Abstracting over HLists

```scala
import shapeless._

object sq extends Poly1 {
  implicit def caseInt = at[Int](x => x * x)
  implicit def caseLong = at[Long](x => x * x)
  implicit def caseFloat = at[Float](x => x * x)
  implicit def caseDouble = at[Double](x => x * x)
  implicit def default[A] = at[A](x => x)
}

def square[L <: HList](implicit
  m: ops.hlist.Mapper[sq.type, L]
)(l: L): L = l map sq
```

We have thrown away the output type, so scalac does not know that m.Out will be equal to L

```
type mismatch;
found   : m.Out
required: L
): L = l map sq
         ^
```

Abstracting over HLists

```
import shapeless._

object sq extends Poly1 {
  implicit def caseInt = at[Int](x => x * x)
  implicit def caseLong = at[Long](x => x * x)
  implicit def caseFloat = at[Float](x => x * x)
  implicit def caseDouble = at[Double](x => x * x)
  implicit def default[A] = at[A](x => x)
}

def square[L <: HList](implicit
  m: ops.hlist.Mapper[sq.type, L] { type Out = L }
)(l: L): L = l map sq
```

We can use a structural refinement type to require that m.Out == L

Abstracting over HLists

```
import shapeless._

object sq extends Poly1 {
  implicit def caseInt = at[Int](x => x * x)
  implicit def caseLong = at[Long](x => x * x)
  implicit def caseFloat = at[Float](x => x * x)
  implicit def caseDouble = at[Double](x => x * x)
  implicit def default[A] = at[A](x => x)
}

def square[L <: HList](implicit
  m: ops.hlist.Mapper.Aux[sq.type, L, L]
)(l: L): L = l map sq
```

Which can be more easily accomplished with the Aux type alias

Generic Representations of Case Classes

- A case class can be represented generically as an HList of its component types — known as the “generic representation”

```
case class Car(make: Make, model: Model, year: Year)
val genericCar: Make :: Model :: Year :: HNil =
  Make("Tesla") :: Model("S") :: Year(2015) :: HNil
```

- Converting a case class to/from its generic representation is accomplished by using Generic

```
trait Generic[T] {
  type Repr
  def to(t : T) : Repr
  def from(r : Repr) : T
}
```

Generic Representations of Case Classes

```
import shapeless.Generic

val car = Car(Make("Tesla"), Model("S"), Year(2015))

val genericCar = Generic[Car]
// genericCar: shapeless.Generic[Car]{type Repr =
shapeless.::[Make,shapeless.::[Model,shapeless.::
[Year,shapeless.HNil]]]} = fresh$macro$12$1@5e21208d

val x = genericCar.to(car)
// x: genericCar.Repr = Make(Tesla) :: Model(S) ::
Year(2015) :: HNil

val y = genericCar.from(Make("VW") :: x.tail)
// y: Car = Car(Make(VW),Model(S),Year(2015))
```

Generic Representations of Case Classes

```
trait Generic[T] {
  type Repr
  def to(t : T) : Repr
  def from(r : Repr) : T
}

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

  def apply[T](implicit gen: Generic[T]): Aux[T, gen.Repr] =
    gen

  implicit def materialize[T, R]: Aux[T, R] =
    macro GenericMacros.materialize[T, R]
}
```

Representation is specified as a type member

Typically need to use structural refinement to make use of this type

Implemented with an implicit macro

From https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/generic.scala

Records

- Generally, a Record is a list of K/V pairs where:
  - the keys are known statically
  - the type of each value is known
- Represented in Shapeless as an HList of a particular shape
  - Each element X_i is represented as FieldType[K_i, V_i] where:

```
type FieldType[K, +V] = V with KeyTag[K, V]
trait KeyTag[K, +V]
```

  - Keys are typically singleton types
    - e.g. Int(23), String("hello")
    - See SIP-23 for more info http://docs.scala-lang.org/sips/pending/42.type.html

Records

```
val car =
  ('make ->> Make("Tesla")) ::
  ('model ->> Model("S")) ::
  ('year ->> Year(2015)) :: HNil

car('make)
// res0: Make = Make(Tesla)

car('model)
// res1: Model = Model(S)

car('year)
// res2: Year = Year(2015)

car + ('year ->> Year(2016))
// res3: … = Make(Tesla) :: Model(S) :: Year(2016) :: HNil
```

Note the result types

Records: Access Non-existent Field

```
val car =
  ('make ->> Make("Tesla")) ::
  ('model ->> Model("S")) ::
  ('year ->> Year(2015)) :: HNil

car('foo)
// <console>:27: error: No field Symbol with
shapeless.tag.Tagged[String("foo")] in record
shapeless.::[Make with shapeless.labelled.KeyTag[Symbol
with
shapeless.tag.Tagged[String("make")],Make],shapeless.::
[Model with shapeless.labelled.KeyTag[Symbol with
shapeless.tag.Tagged[String("model")],Model],shapeless.
::[Year with shapeless.labelled.KeyTag[Symbol with
shapeless.tag.Tagged[String("year")],Year],shapeless.HN
il]]]
```

Type checked
access to fields

Scary!

Records: Full Type

```
val car =
  ('make ->> Make("Tesla")) ::
  ('model ->> Model("S")) ::
  ('year ->> Year(2015)) :: HNil
// car: shapeless.::[Make with
shapeless.labelled.KeyTag[Symbol with
shapeless.tag.Tagged[String("make")],Make],shapeless.::
[Model with shapeless.labelled.KeyTag[Symbol with
shapeless.tag.Tagged[String("model")],Model],shapeless.
::[Year with shapeless.labelled.KeyTag[Symbol with
shapeless.tag.Tagged[String("year")],Year],shapeless.HN
il]]] = Make(Tesla) :: Model(S) :: Year(2015) :: HNil
```

Records: Pretty Printed Type

```
val car =
  ('make ->> Make("Tesla")) ::
  ('model ->> Model("S")) ::
  ('year ->> Year(2015)) :: HNil
// car:
    (Make with KeyTag[
                 Symbol with Tagged[String("make")],
                 Make]) ::
    (Model with KeyTag[
                 Symbol with Tagged[String("model")],
                 Model]) ::
    (Year with KeyTag[
                 Symbol with Tagged[String("year")],
                 Year]) ::
    HNil =
      Make(Tesla) :: Model(S) :: Year(2015) :: HNil
```

Labelled Generic Representations of Case Classes

- A case class can be represented generically as a record of its component fields — known as the “labelled generic representation”

```
case class Car(make: Make, model: Model, year: Year)
val car =
  ('make ->> Make("Tesla")) ::
  ('model ->> Model("S")) ::
  ('year ->> Year(2015)) :: HNil
```

- Converting a case class to/from its labelled generic representation is accomplished by using LabelledGeneric

```
trait LabelledGeneric[T] extends Generic[T]
```

- Same API as Generic — implicit materialization is built off of Generic macro and DefaultSymbolicLabelling type operator

Labelled Generic Representations of Case Classes

```
import shapeless.LabelledGeneric

val car = Car(Make("Tesla"), Model("S"), Year(2015))

val lgenCar = LabelledGeneric[Car]

val x = lgenCar.to(car)
// x: lgenCar.Repr = Make(Tesla) :: Model(S) :: Year(2015)
:: HNil

val y = lgenCar.from(x + ('model ->> Model("X")))
// y: Car = Car(Make(Tesla),Model(X),Year(2015))
```

Labelled Generic: toString with Labels

The generated toString method in case classes does not include field names
https://issues.scala-lang.org/browse/SI-3967

Starting with a concrete example:

```scala
import shapeless.LabelledGeneric
import shapeless.record._

case class Point(x: Int, y: Int, z: Int)

def showPoint(p: Point): String = {
  val rec = LabelledGeneric[Point].to(p)
  rec.fields.toList.map { case (k, v) =>
    s"${k.name}: $v"
  }.mkString("Point(", ", ", ")")
}

showPoint(Point(1, 2, 3))
// res0: String = Point(x: 1, y: 2, z: 3)
```

Labelled Generic: Polymorphic toString with Labels

```scala
import shapeless.{ HList, LabelledGeneric, Typeable }
import shapeless.record._
import shapeless.ops.hlist.ToTraversable
import shapeless.ops.record.Fields

def show[A, R <: HList, F <: HList](a: A)(implicit
  typ: Typeable[A],
  lgen: LabelledGeneric.Aux[A, R],
  fields: Fields.Aux[R, F],
  toList: ToTraversable.Aux[F, List, (Symbol, Any)]
): String = {
  val rec = lgen.to(a)
  rec.fields.toList.map { case (k, v) =>
    s"${k.name}: $v"
  }.mkString(typ.describe + "(", ", ", ")")
}

show(Point(1, 2, 3))
// res1: String = Point(x: 1, y: 2, z: 3)
```

Labelled Generic: Polymorphic toString with Labels

```
def show[A, R <: HList, F <: HList](a: A)(implicit
  typ: Typeable[A],
  lgen: LabelledGeneric.Aux[A, R],
  fields: Fields.Aux[R, F],
  toList: ToTraversable.Aux[F, List, (Symbol, Any)]
): String = {
  val rec = lgen.to(a)
  rec.fields.toList.map { case (k, v) =>
    s"${k.name}: $v"
  }.mkString(typ.describe + "(", ", ", ")")
}

case class Foo[A](value: A)(implicit t: Typeable[A]) {
  override def toString = show(this)
}
show(Foo(1))
// res2: String = Foo[Int](value: 1)
```

Includes parameterized type names!

Shapeless Usage in scodec

HList Codecs

Consider an HList where each component type is a Codec[X_i]

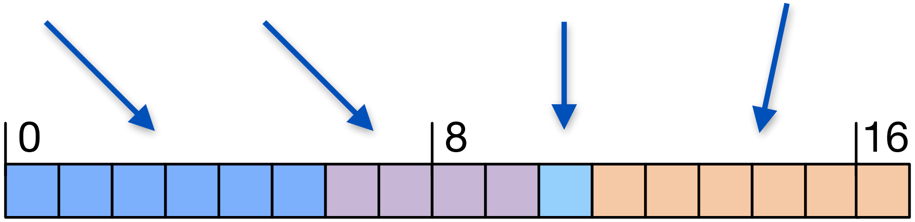
```
import shapeless.{ ::, HNil }
import scodec.codecs._

val cs = int(6) :: ignore(4) :: bool :: int(6) :: HNil

scala> :t cs
Codec[Int] :: Codec[Unit] :: Codec[Boolean] :: Codec[Int] ::
HNil
```

Factoring the Codec type constructor out of the HList gives:

```
Codec[Int :: Unit :: Boolean :: Int :: HNil]
```



HList Codecs

Consider an HList where each component type is a Codec[X_i]

```
import scodec._

val cs = int(6) :: ignore(4) :: bool :: int(6) :: HNil

val ds: Codec[Int :: Unit :: Boolean :: Int :: HNil] =
  cs.toCodec
```

Extension method added via implicit class

```
final implicit class EnrichedHList[L <: HList](self: L) {
  def toCodec(implicit to: codecs.ToHListCodec[L]): to.Out =
    to(self)
}
```

Type operator that implements this conversion

COMCAST

HList Codecs

Combines a Codec[A] and Codec[L <: HList] in to a Codec[A :: L]

```
object HListCodec {
  def prepend[A, L <: HList](
    a: Codec[A], l: Codec[L]
  ): Codec[A :: L] = new Codec[A :: L] {
    override def sizeBound = a.sizeBound + l.sizeBound
    override def encode(xs: A :: L) =
      Codec.encodeBoth(a, l)(xs.head, xs.tail)
    override def decode(buffer: BitVector) =
      Codec.decodeBothCombine(a, l)(buffer) { _ :: _ }
    override def toString = s"$a :: $l"
  }

  object PrependCodec extends Poly2 {
    implicit def caseCodecAndCodecHList[A, L <: HList] =
      at[Codec[A], Codec[L]](prepend)
  }
}
```

Polymorphic function wrapper for prepend

HList Codecs

```
object HListCodec {
  def apply[L <: HList : *->*[Codec]#λ, M <: HList](l: L)(
    implicit folder: RightFolder.Aux[L, Codec[HNil],
                                PrependCodec.type, Codec[M]]
  ): Codec[M] = {
    l.foldRight(hnilCodec)(PrependCodec)
  }

  val hnilCodec: Codec[HNil] = new Codec[HNil] {
    override def sizeBound = SizeBound.exact(0)
    override def encode(hn: HNil) =
      Attempt.successful(BitVector.empty)
    override def decode(buffer: BitVector) =
      Attempt.successful(DecodeResult(HNil, buffer))
    override def toString = s"HNil"
  }
}
```

HList Codecs

```scala
trait ToHListCodec[In <: HList] extends DepFn1[In] {
  type L <: HList
  type Out = Codec[L]
}

object ToHListCodec {
  type Aux[In0 <: HList, L0 <: HList] =
    ToHListCodec[In0] { type L = L0 }

  implicit def mk[I <: HList, L0 <: HList](implicit
    allCodecs: *->*[Codec]#λ[I],
    folder: RightFolder.Aux[I, Codec[HNil],
                    HListCodec.PrependCodec.type, Codec[L0]]
  ): ToHListCodec.Aux[I, L0] = new ToHListCodec[I] {
    type L = L0
    def apply(i: I): Codec[L0] = HListCodec(i)
  }
}
```

HList Codecs: Direct Construction

Rather than building an HList of Codec[X_i], let’s build a Codec[L <: HList] directly

```
import scodec._

val cs = int(6) :: ignore(4) :: bool :: int(6) :: HNil

val ds: Codec[Int :: Unit :: Boolean :: Int :: HNil] =
  int(6) :: ignore(4) :: bool :: int(6)
```

By removing the final HNil, the :: method is being called on the last Codec[Int]

HList Codecs: Direct Construction

```
implicit class EnrichedHListCodec[L <: HList](self:
Codec[L]) {
  import codecs.HListCodec

  def ::[B](codec: Codec[B]): Codec[B :: L] =
    HListCodec.prepend(codec, self)
}

implicit class EnrichedValueCodec[A](self: Codec[A]) {
  import codecs.HListCodec

  def ::[B](codecB: Codec[B]): Codec[B :: A :: HNil] =
    codecB :: self :: HListCodec.hnilCodec
}
```

HList Codecs: Mapping to Case Classes

A Codec[L <: HList] can be converted to Codec[CaseClass] when L is the generic representation of the Case Class

```
import scodec._

val c = int(6) :: int(6) :: int(6)

case class Point(x: Int, y: Int, z: Int)
val d = c.as[Point]

d.encode(Point(1, 2, 3))
// Attempt[BitVector] = Successful(BitVector(18 bits,
0x0420c))
```

HList Codecs: Mapping to Case Classes

Introduce a type class that abstracts over bidirectional lossy transformations

```
abstract class Transform[F[_]] { self =>

  def exmap[A, B](
    fa: F[A],
    f: A => Attempt[B],
    g: B => Attempt[A]): F[B]

  def xmap[A, B](fa: F[A], f: A => B, g: B => A): F[B] =
    exmap[A, B](fa,
      a => Attempt.successful(f(a)),
      b => Attempt.successful(g(b)))

  def as[A, B](fa: F[A])(implicit
    t: Transformer[A, B]): F[B] = t(fa)(self)
}
```

HList Codecs: Mapping to Case Classes

```
implicit class TransformSyntax[F[_], A](
  val self: F[A])(implicit t: Transform[F]) {
    def as[B](implicit t: Transformer[A, B]): Transform[B] =
      t(self)
}

abstract class Transformer[A, B] {
  def apply[F[_]: Transform](fa: F[A]): F[B]
}

object Transformer {
  implicit def fromGeneric[A, Repr, B](implicit
    gen: Generic.Aux[A, Repr],
    bToR: B =:= Repr, rToB: Repr =:= B
  ): Transformer[A, B] = new Transformer[A, B] {
    def apply[F[_]: Transform](fa: F[A]): F[B] =
      fa.xmap(a => gen.to(a), b => gen.from(b))
  }
}
```

HList Codecs: dropUnits Combinator

The ignore codec is a Codec[Unit]

```
import scodec._

val cs = int(6) :: ignore(4) :: bool :: int(6) :: HNil

val ds: Codec[Int :: Unit :: Boolean :: Int :: HNil] =
  int(6) :: ignore(4) :: bool :: int(6)
```

Which causes a Unit to appear in the HList

```
case class Qux(x: Int, flag: Boolean, y: Int)
ds.as[Qux]
// <console>:28: error: Could not prove that shapeless.::
[Int,shapeless.::[Unit,shapeless.::[Boolean,shapeless.::
[Int,shapeless.HNil]]]] can be converted to/from Qux.
```

HList Codecs: dropUnits Combinator

```
import scodec._

val cs = int(6) :: ignore(4) :: bool :: int(6) :: HNil

val ds: Codec[Int :: Unit :: Boolean :: Int :: HNil] =
  int(6) :: ignore(4) :: bool :: int(6)

val es: Codec[Int :: Boolean :: Int :: HNil] =
  ds.dropUnits

case class Qux(x: Int, flag: Boolean, y: Int)
val q = es.as[Qux]
q.encode(Qux(1, true, 2))
// Attempt[BitVector] = Successful(BitVector(17 bits,
0x04210))
```

Derived Codecs for Case Classes

Can we use all of this machinery to generate a Codec[CaseClass] automatically?

Given a case class with generic representation X_0 :: X_1 :: … :: X_n :: HNil …and implicits Codec[X_0], …, Codec[X_n]

We can generate a Codec[CaseClass], which labels each component codec with the field name

```
case class Point(x: Int, y: Int)
case class Line(start: Point, end: Point)
case class Arrangement(lines: Vector[Line])

import scodec.Codec
import scodec.codecs.implicits._

val arrBinary = Codec.encode(arr).require
```

Derived Codecs for Case Classes

```
object Codec {
  implicit val deriveHNil: Codec[HNil] =
    codecs.HListCodec.hnilCodec

  implicit def deriveProduct[H, T <: HList](implicit
    headCodec: Lazy[Codec[H]],
    tailAux: Lazy[Codec[T]]
  ): Codec[H :: T] =
    headCodec.value :: tailAux.value
}
```

Derived Codecs for Case Classes

```
object Codec {

  implicit def deriveRecord[
    KH <: Symbol,
    VH,
    TRec <: HList,
    KT <: HList
  ](implicit
    keys: Keys.Aux[FieldType[KH, VH] :: TRec, KH :: KT],
    headCodec: Lazy[Codec[VH]],
    tailAux: Lazy[Codec[TRec]]
  ): Codec[FieldType[KH, VH] :: TRec] = {
    val headFieldCodec: Codec[FieldType[KH, VH]] =
     headCodec.value.toFieldWithContext(keys().head)
    headFieldCodec :: tailAux.value
  }
}
```

Derived Codecs for Case Classes

```
object Codec {

  implicit def deriveLabelledGeneric[
    A,
    Rec <: HList
  ](implicit
    lgen: LabelledGeneric.Aux[A, Rec],
    auto: Lazy[Codec[Rec]]
  ): Codec[A] = {
    auto.value.xmap(lgen.from, lgen.to)
  }
}
```

Takeaways

- Shapeless enables generic programming in Scala
    - …and pushes the limits of Scala’s capabilities
- Generic programming provides practical benefits to routine programming problems — even mundane problems like handling binary
- Like many disciplines, there are fundamental techniques that occur and reoccur
    - …gaining experience with these techniques leads to "ah-ha!" moments
- Generic programming in Scala is not limited to Shapeless — e.g., Slick has a powerful HList implementation

COMCAST