The Shape of Shapeless

Joseph Abrahamson <joseph@well-conditioned.com>

Presented at LambdaConf 2018

Shapeless is an advanced Scala library that provides facilities for *generic* and *type-level* programming. These are tools for working over the "shape of data" and open the doors to new levels of abstraction. The result is often a typed API that doesn't require use of runtime reflection or macros.

- Compile-time type introspection without writing macros
- Generic derivation of typeclasses
- Sophisticated typed APIs that don't rely on `Any` and runtime reflection

On the other hand, Shapeless can have a substantial comprehension tax. It introduces new kinds of *heterogenously typed data* such as the `HList` and `Coproduct` as well as a whole host of *type-level programming* techniques needed to discuss heterogenously typed data effectively.

This talk is not intended to give a comprehensive overview of what's available in Shapeless. The idea is much more to just give a taste, to sketch the outlines, and to give you a framework for understanding what's going on and how to learn more should you need to.

I want you to walk out of this talk feeling comfortable with the rough ideas that make Shapeless work. I want to help you understand the basic vocabulary for working with it and to show you a few common idioms.

I want you to leave having a sense for the *shape of Shapeless*.

To be prepared for this talk, you should download the `shape-of-shapeless` repo which provides the code for exercises and gives you a development environment with the appropriate dependencies available. Further, you should load this repo up in your Scala development environment and ensure that the dependencies are resolved.

http://github.com/tel/shapeOfShapeless

The Shape of Shapeless

The shape of Shapeless is composed of a few major ideas:

1. the introduction of new types that store rich information about their values, the most canonical being the `HList`
2. the use of type-level programming to interact with these rich types in a way similar to how to we interact with values all at compile-time, and
3. a very minimal set of primitives (defined behind the scene using macros) which expose the structure of case classes (and sealed traits) as rich types introduced in (1) and exploited using (2).

This talk aims to explore 1, 2, and 3 in a brief way and demonstrate how they work together.

What is generic programming?

The problem of *generic programming* is to write code that works over the common patterns of data types. Typically, when you define a new type (via a `class` or `trait` definition) it is something you think of as *opaque* and accessible only via its defined methods.

On the other hand, case classes suggest a different way of looking at types: they're just named collections of subparts. To make this work in Scala the case class machinery defines constructors, pattern matching, extractors, the `copy` method and equality on case classes to all be aligned together. This *vastly* simplifies our image of types and asks the question:

If all of our types are just named collections of subparts, can we write code which exploits this common structure?

It's worth noting right away that runtime reflection offers essentially the same generic view of types that we'll be exploring with Shapeless. So why in the world would you ever use Shapeless over runtime reflection?

Worse, using Shapeless requires the use of several techniques which are very unfamiliar to most users. It can create confusing function signatures and lead to difficult to understand compilation bugs.

The secret is in the names, of course. Runtime reflection happens at *runtime* and when it fails will give you a runtime exception. Shapeless lets us handle our data in a generic way and to do it all during compilation. Use Shapeless for the same reason you use types over, say, contracts. A generic program founded on Shapeless won't even compile when it's wrong.

For instance, Scala automatically provides equality for case classes, but can we write an `Ordering` instance which defers to the `Ordering` of all the sub-components of a case class? The definition feels obvious: it's the same one that we have on `Tuples`, but if you look in the Scala code you see that defining that instance requires writing 22 implementations, one for each arity of `Tuple`.

```
implicit def Tuple2[T1, T2](implicit ord1: Ordering[T1], ord2:
Ordering[T2]): Ordering[(T1, T2)] =
  new Ordering[(T1, T2)]{
    def compare(x: (T1, T2), y: (T1, T2)): Int = {
      val compare1 = ord1.compare(x._1, y._1)
      if (compare1 != 0) return compare1
      val compare2 = ord2.compare(x._2, y._2)
      if (compare2 != 0) return compare2
      0
    }
  }
```

This works so long as (a) we know all of the data types we'd like to support and (b) we're willing to write large amounts of repetitive code. What we'd really like to do is exploit the repeating structure here (defer to orderings on the subcomponents, compare tuples lexicographically) and write just *one* definition that handles all tuples (and all case classes) at once.

By the end of this course, we will write just that definition. Further, we will develop the techniques to write these sorts of generic implementations for any *generic* functionality you dream up.

Generic programming is also a gateway to type-level programming. In order to achieve generic programming we will develop techniques to traverse and react to *structured types*. These allow you to make types that are *vastly more informative* than just opaque names. While we won't get much of a chance to explore this here, this technology opens doors to entirely new kinds of type safety.

A popular example that's been implemented using similar functionality in Haskell is a type-level description of a RESTful API. The entire API description is itself a type (think "Swagger document as type") and type-level programming is used to transform that API description type into the type of a necessarily API-conformant server or autogenerated code for an idiomatic Javascript client.

Generally, these technologies begin to touch on the full power of *dependently typed languages*. Languages which support and facilitate the use of *dependent types* often allow the embedding of arbitrarily complex predictes into types and the creation of large, verified code bases. At the furthest end these languages are used as *proof assistants* used to write mechanically verified proofs of mathematical statements. For instance, the proof of the 4-color theorem was carried out in a dependently typed language.

Generalizing tuples with HLists

We begin exploring Shapeless with the *heterogenous list*. This should at once be very familiar and very strange.

A heterogenous list is a sequence of values of different types. If you try to construct a normal list out of elements of various types you'll end up with a `List[Any]`, a list of a single type that is the supertype of all of the elements. This means that we've actually forgotten the types of the things in our list. A heterogenous list works differently.

You're already familiar with a limit form of HLists: Tuples. A Tuple of size N contains N different component types in order. We'd like to replicate that idea, but make it work consistently for all sizes of HList.

We can *also* achieve that with just Tuple2 and Unit. Unit, the "empty tuple" represents an empty heterogenous list. Tuple2 represents a "cons". Here are some examples

```
val empty: Unit = ()
def cons[A, B](head: A, tail: B): (A, B) = (head, tail)

val length2: (String, (Int, Unit)) = cons("foo", cons(3, empty))

val length5: (String, (String, (Int, (String, (Int, Unit))))) =
  cons("Hello", cons("world", cons(3, length2)))
```

Hopefully, this makes the pattern clear: start with an empty list represented as Unit and then "cons" on as many elements as you like by gluing the head and tail together as a Tuple2. The types of these results grow longer and longer as the size of the list does, but this makes sense: how else would we retain all of the information about the elements of our list?

In a certain sense this is all we need and exactly what Shapeless' `HList` type is, but as written it's difficult to read and understand that a big sequence of `Tuple2`'s ought to be interpreted as an `HList`, so we define our own.

```
// From core/src/main/scala/shapeless/hlists.scala

package shapeless

// The supertype of all heterogenous lists---often used as a
type-bound
// for descriptive purposes

sealed trait HList

// The empty HList is defined as a sealed trait with a single,
concrete
// implementation. Note that it defines :: using the constructor
for cons.

sealed trait HNil extends HList {
   def ::[Head](head: Head) = shapeless.::(head, this)
}

object HNil extends HNil

// The cons operation is defined as a case class so we get both
the type and the
// constructor at once. For convenience, we define it as an
“infix type
// operator”. Note that the Tail must be another HList as we
expect. This ensures
// HLists are properly formed: an improvement over the Tuple2-
based
// implementation.

final case class ::[+Head, +Tail <: HList](head: Head, tail:
Tail) extends HList
```

This is the complete implementation as far as functionality is concerned. In the actual Shapeless codebase there's also a custom definition of `toString`. You can see that this is nothing more than `Unit` and `Tuple2` again with the added constraint that they both belong to the HList supertype. The same examples from before look much nicer now

```
val length2: String :: Int :: HNil = “foo” :: 3 :: HNil
val length5: String :: String :: Int :: String :: Int :: HNil =
  “Hello” :: “world” :: 3 :: length2
```

Notice, importantly, that both `HNil` and `::` show up at both the type and value levels. Take a careful look at the code listing above to notice how this works out.

While we've defined a simple, recursive structure to hold a heterogenous list, we'll quickly find ourselves in a bind: we can't actually do much with them.

We can write simple functions like `length` just fine

```
def hlistLength[L <: HList](value: L): Int = value match {
  case HNil => 0
  case h :: t => 1 + hlistLength(t)
}
```

But all functions we can define in a conventional way share one property: they have to completely disregard all of the information stored in the HList type. So, we've totally defeated the purpose of the HList.

```
// We can turn an HList into a List[Any], which is just another
way of saying
// we've thrown away all of the interesting type information

def toAnyList[L <: HList](value: L): List[Any] = value match {
   case HNil => Nil
   case h :: t => h :: toAnyList(t)
}
```

In order to exploit information about the HList we'll need to dig into type-level programming.

First steps into type-level programming

In order to take advantage of the extra information stored in our HList types we need to write code that does different things at different types: the *values* need to depend upon the *types*. Fortunately, Scala has a mechanism for this: *implicits*. That said, we'll be using them somewhat differently than you might expect.

We'd like to write functions on `HLists` which are sensitive to the types stored in the `HList`. We'll build these functions the same ways we build `HLists`: we start with simple pieces and compose them together in such a way that the resulting type is rich enough to handle what we need.

We'll begin doing this manually and then later pass the process off to the Scala compiler using *implicit resolution*.

Let's begin by writing a function `speak` which transforms `Int`s into `String`s in an `HList`.

```
speak("foo" :: "bar" :: 3 :: 2 :: 1 :: Seq(1, 2, 3) :: HNil)
// => "foo" :: "bar" :: "3" :: "2" :: "1" :: Seq(1, 2, 3) :: HNil
```

This is not a particularly useful function, but it will give us an example of type-level programming. It's also an example of a type-level *map* function—an obervation we'll take advantage of later.

The technique for building type-level functions is the same as what we did with `HLists`: we construct them piece by simple piece in such a way that the types reflect exactly what is happening. In this case, we're building a function that instantiates the following trait.

```
trait Speak[-In <: HList] {
  type Out <: HList
  def apply(in: HList): Out
}
```

We'd like to construct values of that type for exactly the right values of `In` and `Out` to satisfy the specification for how `speak` works.

Another way to think about this is that we will build a small language for constructing values of type `Speak[In]` such that all values created using that language are valid `speak` functions for proper choices of `In` and `Out`.

Before we start defining this language, it's important to talk about the difference between the `In` and `Out` types in `Speak`. Why is `In` a type parameter and `Out` a type member?

The short answer is that it's just a heuristic: when defining type-level functions declare input types as type parameters and output types as type members.

The longer answer is that this constrains the space of possible Speak instances and makes inference work better. In particular, if you have a particular `L <: HList` you can ask me to give you a value `s: Speak[L]`. Once you have it, you can figure out the output type as `s.Out <: HList`. Let's say we instead defined `Speak` as follows

```
trait BadSpeak[-In <: HList, +Out <: HList] extends (In => Out)
```

Now, in order to ask me for a value of `Speak[L, _]` for your given `L`, you have to *already* know what the output type is.

We'll begin simply: we know that there's only one possible behavior for when the input list is the empty `HList`, `HNil`: it must itself return `HNil`.

```
val SpeakHNil: Speak[HNil] = new Speak[HNil] {
  type Out = HNil
  def apply(in: HNil): HNil = in
}
```

This case works as our base case for recursion. All `HLists` are finite so they eventually end up with `HNil`.

Next we'll handle the recursive cases which will let us consider non-empty lists. There are two cases: the type at the head of the `HList` might be an `Int` or it might not. We'll handle each one separately.

The logic here starts out a little weird: in order to handle these more complex cases we'll make a pair of `Speak`-transformer functions. We'll take an input `Speak` instance and *augment* it to handle an `HList` one element longer.

```
def SpeakInt[Tail <: HList](tail: Speak[Tail]): Speak[Int ::
Tail] =
  new Speak[Int :: Tail] {
    type Out = String :: tail.Out
    def apply(in: Int :: Tail): Out = in.head.toString ::
tail(in.tail)
  }
```

Note here that we define the `Out` type member in terms of the "previous" `Speak` instance's `Out` type member.

For the non-`Int` case, we add an extra type argument to stand-in for whatever type we're just ignoring.

```
def SpeakNonInt[Head, Tail <: HList](tail: Speak[Tail]):
Speak[Head :: Tail] =
  new Speak[Head :: Tail] {
    type Out = Head :: tail.Out
    def apply(in: Head :: Tail): Out = in.head :: tail(in.tail)
  }
```

With all of these cases designed, we have a small DSL for building values of Speak for whatever input type we want. For instance, we could define some example list

```
type ExList = String :: String :: Int :: Int :: Seq[Int] :: HNil

val exList: ExList = “foo” :: “bar” :: 3 :: 2 :: Seq(3, 2, 1) ::
HNil

val exListSpeak: Speak[ExList] =

SpeakNonInt(SpeakNonInt(SpeakInt(SpeakInt(SpeakNonInt(SpeakHNil))
)))
```

And with this, we’re *done*! We can apply `exListSpeak` to `exList` and receive an `exListSpeak.Out` as a result.

```
val out: exListSpeak.Out = exListSpeak(exList)
```

Or, really, we’re not quite done at all. There are at least two major issues:

1. Though this all looks fine and compiles, we don’t actually want `out` to have some opaque type `exListSpeak.Out`—we want it to be some known `HList`. In particular, we would expect `head.out` to be of type `String`, but it’s not—it’s not even defined.
2. Nobody in their right mind wants to write out values like `exListSpeak` by hand. Or, even if we did we wouldn’t want to write one for every single `HList` argument we every want to apply `Speak` to. This doesn’t scale.

But despite those two issues, we’ve laid the proper groundwork for type-level programming. We’ll make two small changes next to resolve these issues and produce the *real* definition of `Speak`.

Exercise 1 Write a values of `Speak` that work for each the following HLists

- `true :: false :: true :: 3 :: HNil`
- `3 :: 2 :: 1 :: HNil`
- `true :: (3 :: 2 :: 1 :: HNil) :: HNil`

Exercise 2 The way we've defined `Speak` so far has another issue: it's possible to use our "DSL for constructing `Speak` values" to build one that has invalid behavior. How can you do that?

Improving Speak with implicits

What you're probably thinking at this point is *great—we can now do type-level programming and all, but doing so is such a complete PITA that we never will*.

So far, all our type-level work was done by hand and was completely mechanical. If we know some type `L <: HList` then we can design an algorithm for constructing the appropriate `Speak[L]` value—so why not make Scala make those values for us?

As it turns out, we can do exactly that using *implicits*.

Scala implicits or "Scala's implicit resolution system" is a means for having the compile create values for you at compile time based on their type. You either demand an implicit argument or use the `implicitly[T]` method and the compiler will, at compile time, attempt to construct a value of type `T` for you according to the implicit resolution rules.

These rules are, in generality, quite complex. We'll just use a couple of them but in a very particular way:

- Implicits defined in a type's companion object are always searched. Therefore, we'll define all our implicits in the companion objects of our traits (or traits they inherit from).
- Implicits defined in a trait inherited by a companion object are searched *after* the definitions in the companion object. This lets us define a *priority ordering* necessary later.
- Implicit `def`s which have only `implicit` args will satisfy an implicit search if (a) their return type is what we're looking for and (b) we can find implicit values for all of their arguments. This lets us write *multi-step implicit resolution systems.*

Using these three principles of implicit resolution, we'll make it so that you can ask the Scala compiler to build values of `Speak[L]` implicitly.

The first pass is really simple: we just put all of our `Speak` cases into the companion and mark them `implicit`.

```
object Speak {

  implicit val SpeakHNil: Speak[HNil] = new Speak[HNil] {
    type Out = HNil
    def apply(in: HNil): HNil = in
  }

  implicit def SpeakInt[Tail <: HList](
      implicit tail: Speak[Tail]): Speak[Int :: Tail] =
    new Speak[Int :: Tail] {
      type Out = String :: tail.Out
      def apply(in: Int :: Tail): Out = in.head.toString ::
tail(in.tail)
    }

  implicit def SpeakNonInt[Head, Tail <: HList](
      implicit tail: Speak[Tail]): Speak[Head :: Tail] =
    new Speak[Head :: Tail] {
      type Out = Head :: tail.Out
      def apply(in: Head :: Tail): Out = in.head :: tail(in.tail)
    }

}
```

Now, we can write a more natural interface to this code. We can make a `speak` method which takes in an argument, requests the proper implicit, and returns the result.

```
def speak[In <: HList](in: In)(implicit ev: Speak[In]): ev.Out =
ev(in)
```

While we're here, however, we'll do one more thing. It's very important during resolution that the `Int` case gets checked before the non-`Int` case since the non-`Int` case *covers* the `Int` case. As it turns out, Speak works just fine without this but it is very fragile without it. More than that, this helps broadcast our intentions to the readers of our code.

To ensure that the `Int` case gets resolved first we'll put the non-`Int` case into a `Speak_LowPrio` trait and have the companion inherit it. Implicit resolution always searches direct definitions before inherited ones, so this clearly marks the non-`Int` case as a fallback. The final code looks like

```
trait Speak[-In <: HList] {
  type Out <: HList
  def apply(in: In): Out
}

trait Speak_LowPrio {

  implicit def SpeakNonInt[Head, Tail <: HList](
      implicit tail: Speak[Tail]): Speak[Head :: Tail] =
    new Speak[Head :: Tail] {
      type Out = Head :: tail.Out
      def apply(in: Head :: Tail): Out = in.head :: tail(in.tail)
    }

}

object Speak extends Speak_LowPrio {

  def speak[In <: HList](in: In)(implicit ev: Speak3[In]): ev.Out
= ev(in)

  implicit val SpeakHNil: Speak[HNil] = new Speak[HNil] {
    type Out = HNil
    def apply(in: HNil): HNil = in
  }

  implicit def SpeakInt[Tail <: HList](
      implicit tail: Speak[Tail]): Speak[Int :: Tail] =
    new Speak[Int :: Tail] {
      type Out = String :: tail.Out
      def apply(in: Int :: Tail): Out = in.head.toString ::
tail(in.tail)
    }
```

```
}
```

Unfortunately, while this implementation is now clear and working, while it produces the proper return results, it still doesn't provide enough type information to Scala to work conveniently. After you call `speak` on an `HList`, you lose information about its return type.

```
scala> Speak.speak(HNil)
res1: com.jspha.shapeOfShapeless.Speak.SpeakHNil.Out = HNil

scala> Speak.speak(3 :: HNil)
res2: com.jspha.shapeOfShapeless.Speak[Int :: shapeless.HNil]#Out
= 3 :: HNil
```

Notice here that the return values are correct but the return types are not. We get types like `Speak[Int :: HNil]#Out` while we expect types like `String :: HNil`.

Scala has forgotten too much information and we need to get it back. That's our next step.

Improving Speak with refinements

Generally, type members are treated opaquely. If I give you a value `x` and the type of `x` defines a type member `Tpe` then you can access it as `x.Tpe`, but we can't know specifically what `x.Tpe` is. It's just *some, unknown* type.

This is insufficient for type-level programming like `Speak`: we need to know not just that the output of calling `speak` is *some* Scala type. It's not even good enough to know that it is *some* `HList`—we want to know explicitly which `HList` it is.

For that, we need to *refine* our type information.

The thing is that when we define an implicit value of Speak we actually have all of the type information we need. It's just that Scala *forgets* it.

We can retain this type information by using *refinement types*. When we defined `SpeakHNil` we defined a value of type `Speak[HNil]`, but we actually knew slightly more information: we knew that the `Out` type was itself `HNil` as well. If we give `SpeakHNil` a type which reflects all of that information then Scala will use retain that information for our use later.

The proper type to use is

```
val SpeakHNil: Speak[HNil] { type Out = HNil }
```

The braces following the type are a *refinement* of `Speak` setting its `Out` type member to a known value. It produces a totally new type (e.g. `Speak[HNil]` is not the same type as `Speak[HNil] { type Out = HNil }`) and makes our `SpeakHNil` value more informative.

Generally, whenever you define cases for a type-level function you must use refinement types to declare all of the type members.

This is such a common pattern it has a name: the `Aux` type. When using Shapeless it is good form to define a type alias called `Aux` in the trait's companion object. This Aux type is a stand-in for the refinement type pattern that's a bit easier on the eyes.

```
trait Speak[In <: HList] {
  type Out <: HList
  def apply(in: In): Out
}

object Speak {
  type Aux[In <: HList, Out0 <: HList] = Speak[In] { type Out =
Out0 }
}
```

`Aux` is absolutely just a convention and no different than using the refinement type directly. The only reason you use it is to signal to your reader what's really going on (and to make your types look pretty).

With refinement types and `Aux` in hand we can redefine the cases for `Speak` and have them preserve the necessary output type information. All we need to do is update the type ascriptions for each of our implicit `val`s and `def`s to use the `Aux` type.

```
trait Speak[-In <: HList] {
  type Out <: HList
  def apply(in: In): Out
}

trait Speak_LowPrio {

  implicit def SpeakNonInt[Head, Tail <: HList](
      implicit tail: Speak[Tail]): Speak.Aux[Head :: Tail,
Head :: tail.Out] =
    new Speak[Head :: Tail] {
      type Out = Head :: tail.Out
      def apply(in: Head :: Tail): Out = in.head :: tail(in.tail)
    }

}

object Speak extends Speak_LowPrio {

  type Aux[In <: HList, Out0 <: HList] = Speak[In] { type Out =
Out0 }

  def speak[In <: HList](in: In)(implicit ev: Speak[In]): ev.Out
= ev(in)

  implicit val SpeakHNil: Speak.Aux[HNil, HNil] = new Speak[HNil]
{
    type Out = HNil
    def apply(in: HNil): HNil = in
  }

  implicit def SpeakInt[Tail <: HList](implicit tail:
Speak[Tail])
    : Speak.Aux[Int :: Tail, String :: tail.Out] =
    new Speak[Int :: Tail] {
      type Out = String :: tail.Out
      def apply(in: Int :: Tail): Out = in.head.toString ::
tail(in.tail)
    }

}
```

Now, `Speak` performs exactly as we'd hope. In particular, the *actual* values of `Speak` returned by our implicit resolution are values of `Speak.Aux` for which the `Out` type member is known. Even IntelliJ is perfectly capable of inferring these return types.

```
  type ExList = String :: String :: Int :: Int :: Seq[Int] ::
HNil
  type OutList = String :: String :: String :: String ::
Seq[Int] :: HNil

  val exList: ExList = "foo" :: "bar" :: 3 :: 2 :: Seq(3, 2,
1) :: HNil

  // This works now!
  val out: OutList = speak(exList)

  // IntelliJ can even infer it! Though, it's ugly…
  val outInferred: ::[String, ::[String, ::[String, ::[String, ::
[Seq[Int], HNil]]]] = speak(exList)
```

Recap

At this point, `Speak` is a properly-defined type-level function on `HLists`. It is robust and works perfectly with Scala's inference engine. It uses nice conventions like the `_LowPrio` modules and the `Aux` type. This is everything you should strive for in a type-level function.

Except, of course, it's just a `map` function! Just like at the value level, we don't want to be defining everything by manual recursion. Indeed, map is defined generally in Shapeless in the `shapeless.ops.hlist._` namespace. It's even defined as a method on `HLists` directly if you import the `shapeless.syntax._` namespace

```
def speak[In <: HList](in: In)(implicit mapper:
Mapper[SpeakPoly.type, In]) =
  in.map(SpeakPoly)

object SpeakPoly extends Poly1 {
  val PolyInt = at[Int](_.toString)
  def PolyNonInt[T] = at[T](identity[T])
}
```

Making this works involves figuring out how to use the Shapeless facility for defining polymorphic functions—and it's well worth your time to do so in practice.

But for now, we'll move on to the real meat of Shapeless: the `Generic` typeclass.

Exercise 1 Write a *proper* type-level function that takes the head of an `HList`. Note that it should not work on `HNil`.

Hint You actually only need a single implicit case to make this work. You also don't need to worry about type members or `Aux` types.

Exercise 2 Write a *proper* type-level function which doubles every element in an HList. It should work on all HLists and pass the following tests.

```
doubler(HNil) shouldEqual HNil

doubler(3 :: HNil) shouldEqual 3 :: 3 :: HNil

doubler(true :: 3 :: HNil) shouldEqual true :: true :: 3 :: 3 ::
HNil
```

Hint This one will require type members and output types. Define the `Aux` type for your trait. You won't need to worry about `LowPrio` cases, though.

The Generic typeclass

The real payout of Shapeless is the `Generic` typeclass. It connects the type-level programming techniques over `HLists` we've developed so far to case classes in such a way as to enable compile-time reflection of their structure.

The key use case for this sort of machinery is to write compile-time derived typeclass instances for things like equality and ordering. Throughout this section, we'll use that as a driving example: we'd like to be able to derive lexicographic ordering for any typeclass in a single line.

In particular, we'll use the Generic machinery to write a function `deriveOrdering` which automatically creates an `Ordering` typeclass for any case class we write.

```
final case class Point2d(x: Double, y: Double)

object Point2d {

  implicit val Point2dOrdering: Ordering[Point2d] =
    GenericOrdering.derive[Point2d]

}
```

Moreover, this derivation machinery will even work for recursive types

```
final case class BinTree[A](
    value: A,
    left: Option[BinTree[A]],
    right: Option[BinTree[A]]
)

object BinTree {

  implicit def TreeOrdering[A: Ordering]: Ordering[BinTree[A]] =
    GenericOrdering.derive[BinTree[A]]

}
```

Let's dig in!

Generic is a typeclass, that is a supplemental dictionary of methods related to some type. Let's take a look.

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

Having a value `g: Generic[T]` for some type `T` asserts:

- There is some other type called `g.Repr` which "represents" `T`
- I can transform values of `T` to `g.Repr` with `g.to`
- I can transform values of `g.Repr` back into `T` with `g.from`

This essentially means that `T` and `g.Repr` are "the same", or at least contain the same information.

The reason this is useful is that Shapeless automatically creates instances of `Generic` for all case classes where the `Repr` type is an `HList`. Let's take a look.

We can examine what the auto-generated `Generic` instance is for `Point2d`.

```
scala> final case class Point2d(x: Double, y: Double)
defined class Point2d

scala> import shapeless.Generic
import shapeless.Generic

scala> val g = implicitly[Generic[Point2d]]
g: shapeless.Generic[Point2d] = anon$macro$3$1@2ec30756
```

Instances of `Generic` are produced by macros behind the scenes but you never need to worry about them.

```
scala> g.to(Point2d(2.0, 3.0))
res1: g.Repr = 2.0 :: 3.0 :: HNil
```

By examination we can see that the `g.Repr` type is an `HList`, it's `Double :: Double :: HNil`.

In general, an instance of `Generic` for a case class is an `HList` of all of the case class parameter types listed in order.

The point of `Generic` is that we can translate ideas like "lexicographic ordering" from general implementations on `HLists` to any type with a `Generic` instance (e.g., all case classes). We'll start by implementing a lexicographic ordering on `HLists` and then show how to use `Generic` to extend it to all case classes.

Let's begin by creating an instance of `Ordering` for `HLists` using the techniques we developed earlier.

As before, we begin by writing a trait which implements the function we're interested in defining but is generic in the type of `HList` it takes as arguments. We can use this to build a function that returns `Ordering` instances for `HLists`.

```
trait CompareHLists[L <: HList] {
  def apply(x: L, y: L): Int
}

object CompareHLists {
  def ordering[L <: HList](implicit ev: CompareHLists[L]):
Ordering[L] =
    new Ordering[L] {
      def compare(x: L, y: L): Int = ev(x, y)
    }
}
```

Then, we define the implicit cases for `CompareHLists` in its companion object as well.

```
implicit val CompareHListsHNil: CompareHLists[HNil] =
  new CompareHLists[HNil] {
    def apply(x: HNil, y: HNil): Int = 0
  }

implicit def CompareHListsHCons[H, T <: HList](
    implicit ord: Ordering[H],
    recur: CompareHLists[T]): CompareHLists[H :: T] =
  new CompareHLists[H :: T] {
    def apply(x: H :: T, y: H :: T): Int = {
      val compareHead = ord.compare(x.head, y.head)
      if (compareHead == 0) { recur(x.tail, y.tail) } else 0
    }
  }
```

Note that we have to assume that all of the types within the `HList` have an implicit ordering available.

Next, we'll use `CompareHLists` to compare any type for which we can get a `Generic` instance. Remember, Shapeless automatically provides `Generic` instances for all case classes.

But before we start, it's important to note something I've been inaccurate about: we can't create a lexicographic ordering for *all* case classes. We have `Generic` instances for all case classes, but `CompareHLists` only can be implicitly defined when all of the types in the `HList` have implicit `Ordering` instances available. This will be something Scala handles transparently for us, but it means that our `derive` method will fail—and fail with a terrible error message no less!—whenever we try to derive a case class with component types that cannot be ordered. This is just an unfortunate drawback of Shapeless-based derivations.

The first way we might want to implement our derive method is to ask for the type `T` to have a `g: Generic[T]` instance and then use `g.to` to convert the `T` into an `HList` we could compare using `CompareHLists`. Ultimately, this *is* what we'd like to do, but we won't be able to do it directly. Instead, we'll need to use another `trait`.

```
trait GenericOrdering[T] {
  def apply(x: T, y: T): Int
}

object GenericOrdering {

  def derive[T](implicit ord: GenericOrdering[T]): Ordering[T] =
    new Ordering[T] {
      def compare(x: T, y: T): Int = ord(x, y)
    }

}
```

This is pretty much exactly like our `CompareHLists` trait, but we'll be giving it different implicit cases. In particular, we want a case that works in the following situation:

- Type `T` needs to have an instance `g: Generic[T]`

- Type `g.Repr` needs to be an `HList`
- We need to have an instance `CompareHLists[g.Repr]`

While this idea is straightforward, encoding it into Scala properly requires a small trick, but one we've already seen: the `Aux` type. In particular, the issue is that if we request an implicit value `g: Generic[T]` then there's no way in that same implicit arg-list to reference `g.Repr`.

To resolve this, `Generic` has an `Aux` type which lets you specify both the generic type `T` and its `Repr` type simultaneously. We introduce a new type variable to stand in for `g.Repr` and then constrain the implicit resolution to find a `Generic` instance that fits all of our needs.

```
implicit def GenericOrderingOfCaseClass[T, Repr <: HList](
    implicit g: Generic.Aux[T, Repr],
    ord: CompareHLists[Repr]): GenericOrdering[T] =
  new GenericOrdering[T] {
    def apply(x: T, y: T): Int = ord(g.to(x), g.to(y))
  }
```

—and this is the only case we need to handle! We're done!

Except not really—

In particular, what we've written so far works *just fine* for the `Point2d` example, and it will even `compile` for BinTree examples, but if you actually try to use the instance you'll get a stack overflow due to an infinite loop.

This can be extremely frustrating the first time you run into it!

The issue arises when we request an instance of ordering for `BinTree`. In order to create the instance we need to have Ordering instances for all of the components of BinTree. So, to get a value `implicitly[Ordering[BinTree]]` we need to already have an instance of `implicitly[Ordering[BinTree]]`. Huh?

We can see the issue more clearly if we monomorphize `BinTree` and compare some variations. In particular, we can define `IntBinTree` as follows

```
final case class IntBinTree(
    value: Int,
    left: Option[IntBinTree] = None,
    right: Option[IntBinTree] = None
)

object IntBinTree {

  implicit val TreeOrdering: Ordering[IntBinTree] =
    GenericOrdering.derive[IntBinTree]

}
```

This definition will work without modification. In particular, since `TreeOrdering` is a `val` it will get initialized just once and break the loop. If we change it to a `def` however, we'll end up in the same infinite loop that plagues `BinTree[Int]`.

The essential issue is that while the implicits can be resolved, these methods have to actually be executed at runtime when creating the implicitly requested `Ordering` value. Since everything is defined as `def`s this leads to an infinite loop

at runtime. We need to “tie the knot” somehow to prevent the needless infinite regress.

As it turns out this sort of infinite recursive regress is a common problem with Shapeless. The case we've run into here is classic, but you also frequently will see this as a *compile time* error where `scalac` will complain about "diverging implicit expansion". In each case, the issue is that in either *finding* or *creating* an implicit for some recursive type.

Fortunately, Shapeless provides a utility type called `Lazy` which helps you to resolve recursive loops. You can think of it as being defined like

```
final case class Lazy[+T](lazy val value: => T)
```

This is invalid Scala syntax (and `Lazy` itself is made to work using macros), but the idea is that a value of `Lazy[T]` is like a `lazy val` you can pass around. Importantly, (a) once it resolves it caches the resolution and (b) a value of `Lazy[T]` can be made implicitly available without immediately exploring for implicit values of `T`. These two properties allow `Lazy` to break diverging implicit expansions and runtime infinite loops.

In practice, what you tend to do is request implicit values at type `Lazy[T]` instead of just `T` and then access them using the `.value` member. Also, in practice, it can be quite difficult to determine which implicits should be made `Lazy`. In the case of `GenericOrdering`, you can make it work for recursive types by wrapping two implicits in `Lazy`:

```
object GenericOrdering {

  def derive[T](implicit ord: Lazy[GenericOrdering[T]]):
Ordering[T] =
    new Ordering[T] {
      def compare(x: T, y: T): Int = ord.value(x, y)
    }

  implicit def GenericOrderingOfCaseClass[T, Repr <: HList](
      implicit g: Generic.Aux[T, Repr],
      ord: Lazy[CompareHLists[Repr]]): GenericOrdering[T] =
    new GenericOrdering[T] {
      def apply(x: T, y: T): Int = ord.value(g.to(x), g.to(y))
    }
```

```
}
```

Removing either `Lazy` wrapper makes the infinite loop bug return.

While I can provide the solution here, in general finding these points quickly often is a process of guessing-and-checking by adding `Lazy` wrappers on implicits involved in recursive loops. This isn't terrifically satisfactory, but it can be very difficult to think through and diagnose the exact point where a `Lazy` wrapper is needed.

That said, there are general principles. When using this style of type-level programming, it's often beneficial to make the `derive`-like functions require only `Lazy` traits. Further, you can see that another tight loop is formed when `GenericOrderingOfCaseClass` requires `CompareHLists` which requires `Ordering` which requires `GenericOrderingOfCaseClass`—throwing a `Lazy` wrapper into this loop is what breaks it.

Recap

We've now defined a generic implementation of `Ordering` which works for all case classes including recursively (and co-recursively) defined ones. Doing this at a high-level just involves exploiting the type-level information generated by the `Generic` typeclass using our type-level programming skills developed earlier.

That said, we started to encounter some of the big issues lurking in type-level programming when we ran into a runtime infinite loop caused by the implicit resolution algorithm. This sort of problem is often caused by undue strictness and Shapeless's `Lazy` wrapper can help break these infinite loops.

Beyond this there are two major extensions of the `Generic` technology we've used so far that are available in Shapeless and useful to learn.

First, Shapeless offers `Generic` instances for more than just case classes—it also provides `Generic` instances for sealed traits of case classes. This covers the space of "algebraic data types" which is often rich enough to model most if not all of your domain. In order to understand how `Generic` works in this situation, however, we have to investigate the `Coproduct` type which generalizes `HList` as a "finite combination of components" into a "choice from a finite set of components".

Second, Shapeless also offers `LabelledGeneric` which offers not just the types of each component in a case class (or sealed trait of case classes) but also the *names of the fields*. This is, for instance, how generic JSON encoding libraries make the JSON object keys match the names as they appear in the case class definition. An additional technology is needed here in order to encode the string-like names of these fields into the *type* expressed as `LabelledGeneric[T]#Repr`.

In both cases however, the techniques are extensions of the code techniques we've explored already. Truly, the shape of Shapless arises in the combination of type-level programming applied to macro-generated `Generic` instances.

Exercise 1 Create a function which returns the "size" of any type for which a `Generic` instance is available. It's signature looks like:

```
def genericSize[T: Generic]: Int
```

Note that we don't even need an element of this type!

```
scala> genericSize[Point2d]
res1: Int = 2
scala> genericSize[BinTree[Int]]
res2: Int = 3
```

Hint We don't need to know anything about the `Repr` type to do this except that it's an `HList`.

Exercise 2 Implement a generic derivation of *equality* for all case classes. Scala automatically defines equality for case classes using the `equals` method, but what we want instead is an instance of a typeclass

```
trait Eq[T] {
  def eqv(x: T, y: T): Boolean
}
```

Hint This is very similar to the `Ordering` example we did, so use it as a reference, but try not to just copy it.

Exercise 3 Make a function which transforms any type `T` into any type `S` so long as they have the same `Generic` representation type `Repr`.

Hint To even define the function signature you'll need to make your own trait to define the property that `T` and `S` share. For instance, you might have a signature like

```
def interchange[T, S](t: T)(implicit ev: Interchangeable[T, S]):
S
```

At this point it's my hope that you have seen the *shape* of Shapeless. We've discussed the primary techniques and demonstrated how gluing them together enables the technique of compile-time generic programming.

But why would you ever chose to use compile-time generic programming?

The general reason for using generic programming is that by exploiting the structure of data we can avoid writing substantial amounts of boilerplate code. In fact, the original papers where these ideas were explored in a typed setting were named "Scrap Your Boilerplate" as it is a major, driving factor. Programmers who write code in dynamic languages use this sort of structural programming all the time and benefit from very short programs.

In Scala however we have another option: we can use runtime reflection to examine and exploit our types. This can often be much friendlier than the Shapeless techniques we've seen so far. For instance, a function which computes the "size" of a case class can be written as a one-liner using `productIterator`.

The issue with generic programming atop reflection is that you end up throwing away all of your typed guardrails. Reflecting on a type gives you complete control and complete information, but subsequently makes it very easy to make mistakes or to define generic functionality which is only partially defined. For this reason, when working with generic code that is often a bit complicated to reason about on its own, typeful generic programming techniques like Shapeless can be very beneficial.

The other competing technique for generic programming is using Scala's macro facility directly. Shapeless is technically based on macros, but the interface you use (the `Generic` typeclass) hides them entirely. Writing generic code with macros is very fragile both because macros are technically not yet a truly supported Scala feature (!) and because macros abstract over the Scala syntax. This is a much richer, and more complex way of looking at the structure of data

than what `Generic` provides. On the other hand, sometimes you truly need full syntactic abstraction.

In general, consider the end user first, though, when writing generic functionality. How confusing will it be to witness your library's API? Will they have to understand the shape of Shapeless to even begin to use your code? Can you create documentation that helps them get by without as much pain? Will they read it?

Can you create comparable functionality using macros or runtime reflection? Will these techniques create other drawbacks for your users?

Use Shapeless when it creates a big advantage for your users by reducing substantial boilerplate and the errors which go along with such code. Use it to avoid complex runtime reflection or syntactic reflection using macros when those techniques would produce opportunity for user confusion or runtime failure.

Finally, most generally, use Shapeless to understand a bit better how the structure of types relates to the structure of your code. Ultimately, the shape of Shapeless is the shape of dependently typed programming in Scala. Understanding Shapeless will dramatically improve your understanding of Scala's type system—particularly implicit resolution and inference.

There's a lot more to learn about Shapeless. The following links are useful resources for digging in.

- **Shapeless ScalaDocs** are a little difficult to find, but are a good way to browse the code and see how the packages are arranged. <https://javadoc.io/doc/com.chuusai/shapeless_2.12/2.3.3>
- **Shapeless examples** demonstrate many techniques of type-level programming and typically are a good way to figure out what *idiomatic* type-level programming looks like <https://github.com/milessabin/shapeless/tree/master/examples/src/main/scala/shapeless/examples>
- **The Shapeless Gitter channel** is a good place to ask questions about using Shapeless and learn from the community and maintainers. <https://gitter.im/milessabin/shapeless>
- **The Shapeless Wiki** is for the most part a changelog for each new version of Shapeless. You can view this to get a picture for the features introduced to Shapeless over time. <https://github.com/milessabin/shapeless/wiki>
- **The *Type Astronaut's Guide to Shapeless* by Dave Gurnell** is a solid e-book describing the use of Shapeless for common typeclass derivation techniques. This would be a very good follow-up to this lecture. <https://underscore.io/books/shapeless-guide/>

Joseph Abrahamson is a data scientist and software engineer specializing in functional programming, Bayesian statistics, and startups. He graduated from the Georgia Institute of Technology with a B.S. in Biomedical Engineering, attended Johns Hopkins university for an incomplete PhD in statistics, and is a cofounder of Reify Health and Signal Vine. Currently, he works at CiBO Technologies on building learning systems for improving agricultural systems from the single field to the global ecosystem.

He's interested in how data science can be improved through the use of language theory and how functional programming opens new doors for human inference and learning.

He lives in Cambridge, MA and can be contacted at <joseph@well-conditioned.com>.