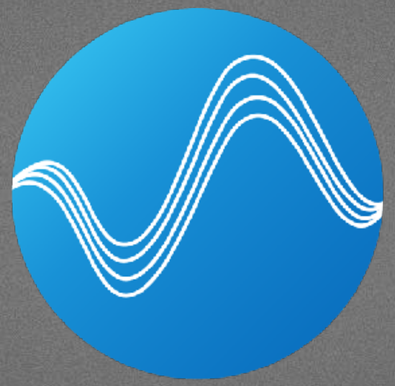


Introduction to type-level (meta)programming with Shapeless

Piotr Krzemiński > ScalaWave 2016

Agenda

- value-level vs. type-level programming
- type-level constraints
- introduction to Shapeless
- abstracting over arity
- typeclasses & typeclass derivation
- metaprogramming alternatives: implicit macros
- advanced Shapeless
- real world use cases

Technical stuff

- Please clone repository:

  https://github.com/krzemin/scalawave-metaprogramming

- You will need SBT and your favorite IDE or text editor

- This presentation is exported as PDF in the repository

Organizational stuff

- I propose one 10-15 min. break
- You will be given few exercises to solve on your own
- Don’t hesitate to ask me you feel you don’t fully understand something

Type-level vs. value-level programming

What is type-level programming?

- value-level programming is describing computations performed on values
- type-level programming is describing computations performed on types

What type-level programming may be used for?

- improve type-safety of your value-level programs:
  - enforce additional constraints that prevents runtime errors by converting them into type errors (catched at compile time)
- abstract over arity
  - overcome limitation of 22 function arguments or 22 fields in case class
- automatic code generation/derivation (a.k.a. boilerplate elimination)
  - by automatic deriving implicit type-class instances for arbitrary case classes

What type-level programming may be used for?

```scala
import io.circe._, io.circe.generic.auto._
import io.circe.parser._, io.circe.syntax._

sealed trait Foo
case class Bar(xs: List[String]) extends Foo
case class Qux(i: Int, d: Option[Double]) extends Foo

val foo: Foo = Qux(13, Some(14.0))

foo.asJson.noSpaces
// res0: String = {"Qux":{"i":13,"d":14.0}}

decode[Foo](foo.asJson.spaces4)
// res1: cats.data.Xor[io.circe.Error,Foo] = Right(Qux(13,Some(14.0)))
```

Warm-up exercise - ex0

implement few operations on Peano Number arithmetics

```
sealed trait Nat
case object Zero extends Nat
case class Succ(prev: Nat) extends Nat

val zero: Nat = Zero
val one: Nat = Succ(Zero)
val three: Nat = Succ(Succ(one))
```

Warm-up exercise - ex0

```
def toInt(n: Nat): Int = {
  ???
}

def add(n1: Nat, n2: Nat): Nat = {
  ???
}
```

- use pattern matching and recursion
- focus on minimal number of dispatched cases
- don’t try to optimize it, don’t need to use tail recursion

Warm-up exercise - `ex0`

- go to `scalawave.exercise.ex0` package
- run `sbt "~testOnly scalawave.exercise.ex0.*"`
- your solution is ready when all your tests pass

Scala features recap

- deep understanding implicits
- type members, path-dependent types
- compile-time recursion on implicits

Implicit scope

<u>implicit scope</u> = mapping from types to values

Implicit scope

<u>implicit scope</u> = mapping from types to values

- at most one value of given type may exists in implicit scope
- usually names don't matter
- introduce value to implicit scope by:

```
implicit val x: T = value
implicit def y: U = value
implicit object z extends V { … }
```

Implicit scope

<u>implicit scope</u> = mapping from types to values

- at most one value of given type may exists in implicit scope
- usually names don't matter
- introduce value to implicit scope by:

```
implicit val x: T = value
implicit def y: U = value
implicit object z extends V { … }
```

implicit scope mapping:

T -> x
U -> y
V -> z

Implicit parameters

defs may have single implicit parameter list

```
def compute(e: Expr)(implicit ctx: Context): Result = {
    // …
    case Add(e1, e2) => compute(e1)(ctx) + compute(e2)(ctx)
    // …
}
```

invoke function without explicit passing Context argument

```
implicit val ctx: Context = Context.empty
compute(Add(x, y))
```

Implicit parameters

implicit parameter is added to implicit scope in `def`'s body

```
def compute(e: Expr)(implicit ctx: Context): Result = {
    // …
    case Add(e1, e2) => compute(e1) + compute(e2)
    // …
}
```

Implicit conversions

since functions are first-class values, we may
import them into implicit scope as well

```
implicit val intToStr: Int => String = _.toString
implicit def doubleToStr(v: Double): String = v.toString

List[String]("well, ", 2, " + ", 3.0, " = ", 5).reduce(_ + _)
// res0: String = well, 2 + 3.0 = 5
```

you are not encouraged to use them
as they may pollute language semantics!

Implicit classes

- special case of implicit conversion
- you can use it to extend your types with new methods
- has only local effect (they are visible only when you import implicit class into scope), does not pollute your original type

```
implicit class MapOps[K, V](mapKV: Map[K, V]) {
  def mapKeys[L](f: K => L): Map[L, V] =
    mapKV.map { case (k, v) => f(k) -> v }
}
```

Implicit classes

```
implicit class MapOps[K, V](mapKV: Map[K, V]) {
    def mapKeys[L](f: K => L): Map[L, V] = …
}
```

is desugared to:

```
implicit def toMapOps[K, V](mapKV: Map[K, V]): MapOps[K, V] =
  new MapOps[K, V](mapKV)

class MapOps[K, V](mapKV: Map[K, V]) {
   def mapKeys[L](f: K => L): Map[L, V] = …
}
```

Context bounds

```
def sort[T: Ordering](xs: List[T]): List[T]
```

is equivalent to:

```
def sort(xs: List[T])(implicit ord: Ordering[T]): List[T]
```

Accessing implicit values

Ok, but how to access implicit `Ordering[T]` when having it declared as context bound?

```
def sort[T: Ordering](xs: List[T]): List[T] = {
  …
  implicit val ord = implicitly[Ordering[T]]
}
```

`implicitly[T]` peeks value of type `T` from implicit scope

Implicit evidences

```
implicitly[T =:= U]
implicitly[T <:< U]
implicitly[T =:!= U] // provided by shapeless
```

```
implicitly[Int =:= Int]
// res0: Int =:= Int = <function1>

implicitly[Int =:= String]
// error: Cannot prove that Int =:= String.
```

Implicit search

during the compilation, when implicit value of type T is requested, type-checker performs implicit search algorithm

1. examine current scope
   - implicits defined in current scope
   - explicit imports
   - wildcard imports
2. if not found, look for implicit in
   - companion objects of a type T
   - suprertypes of companion objects
   - implicit scope of an argument's type
   - implicit scope of type arguments
   - outer objects for nested types

Type members

```
trait Animal {
  type Food
  def breakfast: Food
}
```

```
trait Mouse extends Animal {
  type Food = Cheese
}
```

```
trait Cat extends Animal {
  type Food = Mouse
}
```

Path-dependent types

- generally dependent type is a type whose definition depends on a value
- Scala type system does not embrace fully fledged dependent types
- limited support for dependent types is provided:
  types that may depend on a path

```
val jerry = new Mouse { val breakfast = Gouda }
val tom = new Cat { val breakfast = jerry }

implicitly[jerry.Food =:= Cheese]
implicitly[tom.Food =:= Mouse]
implicitly[tom.breakfast.Food =:= Cheese]
```

Path-dependent types

- generally dependent type is a type whose definition depends on a value
- Scala type system does not embrace fully fledged dependent types
- limited support for dependent types is provided:
  types that may depend on a path

```
val jerry = new Mouse { val breakfast = Gouda }
val tom = new Cat { val breakfast = jerry }

implicitly[jerry.Food =:= Cheese]
implicitly[tom.Food =:= Mouse]
implicitly[tom.breakfast.Food =:= Cheese]
```

Type classes

```
trait Ordering[T] {
  def compare(x: T, y:  T): Int
}
trait Numeric[T] {
  def plus(x: T, y: T): T
  def minus(x: T, y: T): T
  …
}
trait Show[T] {
  def show(x: T): String
}
```

- type class is just an interface parameterized by some type
- you can make concrete type a member of some type class by defining implicit instance

Type classes

```
implicit val intShow: Show[Int] = new Show[Int] {
  def show(x: Int) = x.toString
}

implicit object stringShow extends Show[String] {
  def show(x: String) = x
}

implicit def pairShow[A,B](implicit sa: Show[A], sb: Show[B]): Show[(A,B)] =
  new Show[(A,B)] {
    def show(x: (A,B)) = "(" + sa.show(x._1) + ", " + sb.show(x._2)+ ")"
  }
```

- you can make concrete type a member of some type class by defining implicit instance
- implicit instances may require instances to be defined for other types

Type classes

```
implicit def pairShow[A,B](implicit sa: Show[A], sb: Show[B]): Show[(A, B)] =
  new Show[(A, B)] {
    def show(x: (A, B)) = "(" + sa.show(x._1) + ", " + sb.show(x._2)+ ")"
  }

implicit def pairShow[A: Show, B: Show]: Show[(A, B)] = new Show[(A, B)] {
  def show(x: (A, B)) =
    "(" + implicitly[Show[A]].show(x._1) + ", " +
          implicitly[Show[B]].show(x._2) + ")"
  }
```

- these definitions are equivalent
- implicit parameter list was replaced by context bounds

Type-level Peano Arithmetics - ex1

```
sealed trait Nat
class Zero extends Nat
class Succ[N <: Nat] extends Nat
```

- slightly different representation than in `ex0`
- no `(prev: Nat)` as `Succ` class member
- `Succ` type-parameterized with `N <: Nat`
- we are only interested in types, we won't write `new Zero` or `new Succ[…]` for any reason

Type-level Peano Arithmetics - ex1

```scala
sealed trait Nat
class Zero extends Nat
class Succ[N <: Nat] extends Nat
```

```scala
// Zero is already a type!
type One = Succ[Zero]
type Three = Succ[Succ[Succ[Zero]]]
type Four = Succ[Three]
```

Compile-time recursion

General recipe for type-level recursion on implicits

VALUE-LEVEL	TYPE-LEVEL
recursive function	type class
base case (non recursive value return)	`implicit val` type class instance
recursive function invocation	`implicit def` with implicit parameter required for recursive reference

Compile-time recursion

General recipe for type-level recursion on implicits

VALUE-LEVEL	TYPE-LEVEL

`def toInt(n: Nat): Int`	`trait ToInt[N <: Nat] {`
	`  def value: Int`
	`}`

`case Zero => 0`	`implicit val zeroCase = new ToInt[Zero] {`
	`  val value: Int = 0`
	`}`

`case Succ(k) => 1 + toInt(k)`	`implicit def succCase[K <: Nat]`
	`  (implicit toInt: ToInt[K]) =`
	`  new ToInt[Succ[K]] {`
	`  val value: Int = 1 + toInt.value`
	`  }`

Type error assertions

Sometimes we want to assert that some type-level computation should fail

Idea: write macro that:

- takes code as a string and parse it
- if the code type-checks, then it produce compilation error
- if the code doesn’t type-check, then it succeeds

Type error assertions

Fortunately, we don’t need to write that macro on our own!

shapeless.test.illTyped

```
import shapeless.test._

illTyped("implicitly[Int =:= String]") // this one succeeds

illTyped("implicitly[Int =:= Int]")
// Type-checking succeeded unexpectedly.
// Expected some error.
```

Type-level Peano Arithmetics - `ex1`

- go to `scalawave.exercise.ex1` package
- run `sbt "~testOnly scalawave.exercise.ex1.*"`
- your solution is ready when all your tests pass

Debugging implicits

- `implicitly[T]`
- `shapeless.the[T]`
- `scala.reflect.runtime.universe.reify`
- `-Xlog-implicits`

Debugging implicits

```
scala> implicitly[Add[Succ[Zero], Zero]]
res1: Add[Succ[Zero],Zero] = Add$$anon$4@6f209a

scala> shapeless.the[Add[Succ[Zero], Zero]]
res2: Add[Succ[Zero],Zero]{type Result = Succ[Zero]} = Add$
$anon$4@40bcfd5b

scala> show(reify(implicitly[Add[Succ[Zero], Succ[Zero]]]).tree)
res3: String = Predef.implicitly[Add[Succ[Zero], Succ[Zero]]]
(Add.succCase(Add.zeroCase))
```

@implicitNotFound

```
implicitly[Lt[Zero, Zero]]

error: could not find implicit value for parameter e:
scalawave.ex1.Lt[scalawave.ex1.Zero,scalawave.ex1.Zero]
       implicitly[Lt[Zero, Zero]]
                  ^
```

- compilation error message is not very precise
- we may want to slightly improve it

@implicitNotFound

```
@implicitNotFound("Can't prove that ${N} is less than ${M}")
class Lt[N <: Nat, M <: Nat]
```

- there is special annotation `@implicitNotFound`
- we may provide own compilation error message template
- we may access type parameters
- watch out to not introduce string interpolation syntax!

@implicitNotFound

```
implicitly[Lt[Zero, Zero]]

error: Can't prove that Zero is less than Zero
       implicitly[Lt[Zero, Zero]]
                 ^
```

- our own compilation message is printed when implicit instance was not found
- it’s not perfect, but may guide user where is exact reason why the code doesn’t compile

Type-level constraints

Compile-time invariant checking

- we want our programs to be correct
- we use types to ensure our confidence about correctness of written code
- still there are some places, where we can get runtime errors

```
Nil.head
// java.util.NoSuchElementException: head of empty list

List(1, 2, 3).apply(5)
// java.lang.IndexOutOfBoundsException: 5
```

Compile-time invariant checking

Idea:

- define collection type, that maintains type-level information about its size `N <: Nat`
- every operation like `::` or `++` should update the size information
- `head` on empty list may produce compile-time error
- `at[K <: Nat]` that ensures bounds

Sized Vector - `ex2`

- go to `scalawave.exercise.ex2` package
- run `sbt "~testOnly scalawave.exercise.ex2.*"`
- your solution is ready when all your tests pass

Abstracting over arity

Sized Vector - missing constructor

For ordinary `List[T]` we have convenient constructor:

```
List(1, 2, 3) // : List[Int]
```

```
object List {
  def apply[T](elems: T*): List[T]
}
```

Sized Vector - missing constructor

We would want to define similar constructor for our sized `Vector[N, T]` so that:

```
Vector(1, 2, 3) // : Vector[Succ[Succ[Succ[Zero]]], Int]
```

```
object Vector {
  def apply[T](elems: T*): Vector[?, T]
}
```

we can’t reconstruct type information about length

Sized Vector - missing constructor

We may try direct approach

```
object Vector {
  def apply[T]():  Vector[Zero, T] = VNil
  def apply[T](e1: T): Vector[Succ[Zero], T] = e1 :: VNil
  def apply[T](e1: T, e2: T): Vector[Succ[Succ[Zero]], T] =
    e1 :: e2 :: VNil
  // and so on …
}
```

Abstracting over arity

With type-level programming and few shapeless abstractions we can define single `apply[T](…): Vector[N, T]` that can take arbitrary* number of arguments and compute its length.

But first we need to introduce some shapeless basics.
Stay tuned.

Introduction to Shapeless

- generic programming library
- created and maintained by Miles Sabin since 2011
- extensively uses type classes and path-dependent types
- provides useful abstractions for type-level programming
- very close to being standard library for type-level computations
- pushes to the limits what is possible in Scala

Product and coproduct types

Product examples:

```
(Int, String)
case class User(id: Int, name: String, height: Double)
List[T]
```

Coproduct examples:

```
sealed trait Color
class Red extends Color
class Green extends Color
class Blue extends Color
Either[Error, Result]
```

Product and coproduct types

`Products` are conjunction types,
we hold few things at the same time

`Coproducts` are union/disjunction types,
we hold one thing of few possible alternatives

Product types

Scala product types limitations:

- maximum number of 22 members
- they are defined directly, they don’t have recursive structure
- hard to define recursive computations on them

Shapeless product type - HList

```scala
sealed trait HList

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

Shapeless product type - HList

```
import  shapeless._

val foo = "abc" :: 42 :: false :: HNil
foo: String :: Int :: Boolean :: HNil = abc :: 42 :: false :: HNil

val bar = 123.321 :: foo
bar: Double :: String :: Int :: Boolean :: HNil = 123.321 :: abc ::
42 :: false :: HNil
```

They might be considered as arbitrarily-sized tuples

Shapeless product type - HList

```
import  shapeless._

def hlistLength(hlist: HList): Int = hlist match {
  case HNil => 0
  case _ :: tail => 1 + hlistLength(tail)
}

hlistLength(foo) // 3 : Int
hlistLength(bar) // 4 : Int
```

Shapeless product type - HList

```
import  shapeless._
import  shapeless.ops.hlist._

bar.toList
// List(123.321, abc, 42, false) : List[Any]

val baz = 1 :: 2 :: 3 :: HNil
// baz: Int :: Int :: Int :: HNil = 1 :: 2 :: 3 :: HNil

baz.toList
// List(1, 2, 3) : List[Int]
```

Shapeless product type - HList

```
trait ToList[L <: HList, Lub] {
  def apply(hlist: L): List[Lub]
}

implicit class HListOps[L <: HList](hlist: L) {
  def toList[Lub](implicit toList: ToList[L, Lub]): List[Lub] =
      toList.apply(hlist)
}
```

Shapeless Generic

- `HList` has some cool properties, but we can't force people to use it directly
- people want to use standard scala constructs like case classes or tuples
- `Generic[T]` is typeclass for convenient converting between standard scala products and shapeless representation

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

Shapeless Generic

```
case class Car(brand: String,
               model: String,
               productionYear: Int)
```

to: Car => Repr

from: Repr => Car

```
type Repr = String :: String :: Int :: HNil
```

Shapeless Generic

```
val carGen = implicitly[Generic[Car]]
// Generic[Car] { type Repr = String :: String :: Int :: HNil }

val c = Car("Cadillac", "DeVille", 1961)

val hc = carGen.to(c)
// Cadillac :: DeVille :: 1961 :: HNil  :  String :: String :: Int :: HNil

val c2 = carGen.from(hc)
// Car(Cadillac, DeVille, 1961)  :  Car

val maluch = carGen.from("Fiat" :: "126P" :: 1976 :: HNil)
// Car(Fiat, 126P, 1976)  :  Car
```

Shapeless Generic

- implemented with macros

```
object Generic {
  implicit def apply[T]: Generic[T] = macro ...
}
```

- **compile-time reflection** (don't confuse with runtime reflection!)
- **both target** `Repr` **type and methods** `to: T => Repr` **and** `from: Repr => T` **are generated**

Shapeless Generic

```
val carGen = new Generic[Car] {
  type Repr = String :: String :: Int :: HNil

  def to(car: Car): Repr = {
    car.brand :: car.model :: car.productionYear :: HNil
  }

  def from(repr: Repr): Car = repr match {
    case brand :: model :: productionYear :: HNil =>
      Car(brand, model, productionYear)
  }
}
```

Path dependent types & implicit params

```
trait HListLength[L <: HList] {
  type Length <: Nat
}

def compute[T](obj: T)
              (implicit gen: Generic[T],
                        len: HListLength[gen.Repr]): len.Length = ???
```

Path dependent types & implicit params

```
trait HListLength[L <: HList] {
  type Length <: Nat
}

def compute[T](obj: T)
              (implicit gen: Generic[T],
                        len: HListLength[gen.Repr]): len.Length = ???

illegal dependent method type: parameter may only be referenced in a
subsequent parameter section

Compilation Failed
```

Path dependent types & implicit params

```
trait HListLength[L <: HList] {
  type Length <: Nat
}

def compute[T](obj: T)
              (gen: Generic[T])
              (len: HListLength[gen.Repr]): len.Length = ???
```

Path dependent types & implicit params

```
trait HListLength[L <: HList] {
  type Length <: Nat
}

def compute[T](obj: T)
              (gen: Generic[T])
              (len: HListLength[gen.Repr]): len.Length = ???

type arguments [gen.Repr] do not conform to trait HListLength's type
parameter bounds [L <: shapeless.HList]

Compilation Failed
```

Partial solution: type refinements

```
trait HListLength[L <: HList] {
  type Length <: Nat
}

def compute[T](obj: T)
              (gen: Generic[T] { type Repr <: HList })
              (len: HListLength[gen.Repr]): len.Length = ???
```

Solution: additional type parameter

```
trait HListLength[L <: HList] {
  type Length <: Nat
}

def compute[T, L <: HList](obj: T)
                          (implicit gen: Generic[T] { type Repr = L },
                           len: HListLength[L]): len.Length = ???
```

Refactor: Aux type alias

```
trait HListLength[L <: HList] {
  type Length <: Nat
}

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

def compute[T, L <: HList](obj: T)
                          (implicit gen: Generic.Aux[T, L],
                           len: HListLength[L]): len.Length = ???
```

Refactor: Aux type alias

```
trait HListLength[L <: HList] {
  type Length <: Nat
}

object HListLength {
  type Aux[L, N] = HListLength[L] { type Length = N }
}

def compute[T, L <: HList, N <: Nat](obj: T)
                                     (implicit gen: Generic.Aux[T, L],
                                     len: HListLength.Aux[L, N]): N = ???
```

Refactor: Aux type alias

```
trait HListLength[L <: HList] {
  type Length <: Nat
}

object HListLength {
  type Aux[L, N <: Nat] = HListLength[L] { type Length = N }
}

def compute[T, L <: HList, N <: Nat](obj: T)
                                     (implicit gen: Generic.Aux[T, L],
                                     len: HListLength.Aux[L, N]): N = ???
```

Abstracting over arity - `ex3`

Idea for implement Vector.apply of arbitrary arity

- take `P <: Product` as input parameter
- convert it to generic representation `L <: HList`
- compute `N <: Nat` - length of `L` encoded on type-level
- convert `L` to ordinary scala `List[T]`
- construct `Vector[N, T]` of demanded size

Abstracting over arity - `ex3`

- go to `scalawave.exercise.ex3` package
- run `sbt "~testOnly scalawave.exercise.ex3.*"`
- your solution is ready when all your tests pass

Abstracting over arity - quirks

Yeah, abstracting over arity is cool example of what we can type-level programming to, but it has some problems:

- we have to define separate function for empty and unary instances
- `Vector(Car("Fiat", "126P", 1976)) == "Fiat" :: "126P" :: 1976 :: VNil // WAT?`
- `Vector(Car("Fiat", "126P", 1976), Car("Fiat", "126P", 1976)) ==`
  `Car("Fiat", "126P", 1976) :: Car("Fiat", "126P", 1976) :: VNil`

Type class derivation

Type class derivation - motivation

```
import io.circe._, io.circe.generic.auto._
import io.circe.parser._, io.circe.syntax._

sealed trait Foo
case class Bar(xs: List[String]) extends Foo
case class Qux(i: Int, d: Option[Double]) extends Foo

val foo: Foo = Qux(13, Some(14.0))

foo.asJson.noSpaces
// res0: String = {"Qux":{"i":13,"d":14.0}}

decode[Foo](foo.asJson.spaces4)
// res1: cats.data.Xor[io.circe.Error,Foo] = Right(Qux(13,Some(14.0)))
```

Type class derivation - motivation

```scala
sealed trait JsValue

case object JsNull extends JsValue
case class JsNum(num: BigDecimal) extends JsValue
case class JsStr(str: String) extends JsValue
case class JsBool(bool: Boolean) extends JsValue

case class JsArr(elems: List[JsValue]) extends JsValue
case class JsObj(fields: List[(String, JsValue)]) extends JsValue
```

Type class derivation - motivation

```
def toString(value: JsValue): String = value match {
  case JsNull => "null"
  case JsNum(num) => num.toString()
  case JsStr(str) => s""""$str""""
  case JsBool(bool) => bool.toString
  case JsArr(elems) =>
    s"[${elems.map(toString).mkString(", ")}]"
  case JsObj(fields) =>
    s"{${fields.map(fieldToString).mkString(", ")}}"
}

def fieldToString(field: (String, JsValue)) =
  s""""${field._1}": ${toString(field._2)}"""
```

Type class derivation - motivation

```
trait JsonEncoder[T] {
  type Out <: JsValue
  def toJson(obj: T): Out
}

implicit class JsonOps[T](val obj: T) extends AnyVal {
  def toJson(implicit enc: JsonEncoder[T]): enc.Out =
    enc.toJson(obj)
}
```

Type class derivation - motivation

```scala
trait JsonEncoder[T] {
  type Out <: JsValue
  def toJson(obj: T): Out
}

implicit val intInstance: JsonEncoder[Int] = ...
implicit val stringInstance: JsonEncoder[String] = ...
implicit val boolInstance: JsonEncoder[Boolean] = ...

5.toJson // == JsNum(5)
"Abc".toJson // == JsStr("Abc")
false.toJson // == JsBool(false)
```

Type class derivation - motivation

```
case class Address(country: String,
                   city: String,
                   street: String,
                   postalCode: String)

implicit val addressInstance: JsonEncoder[Address] =
  new JsonEncoder[Address] {
    type Out = JsObj
    def toJson(addr: Address): Out = JsObj(
      "country" -> addr.country.toJson,
      "city" -> addr.city.toJson,
      "street" -> addr.street.toJson,
      "postalCode" -> addr.postalCode.toJson
    )
  }
```

Type class derivation - motivation

```
case class Address(country: String,
                   city: String,
                   street: String,
                   postalCode: String)

val scalacAddr =
  Address("Poland", "Gdańsk", "Bora-Komorowskiego 19B", "80-377")

scalacAddr.toJson == JsObj(
  "country" -> JsStr("Poland"),
  "city" -> JsStr("Gdańsk"),
  "street" -> JsStr("Bora-Komorowskiego 19B"),
  "postalCode" -> JsStr("80-377")
)
```

Type class derivation - motivation

```
case class Location(latitude: Double,
                    longitude: Double)

implicit val locationInstance: JsonEncoder[Location] =
  new JsonEncoder[Location] {
    type Out = JsObj
    def toJson(loc: Location): Out = JsObj(
      "latitude" -> loc.country.toJson,
      "longitude" -> loc.city.toJson
    )
  }
```

Type class derivation - motivation

```
case class Location(latitude: Double,
                    longitude: Double)

val scalacLocation = Location(54.409472, 18.584570)

scalacLocation.toJson == JsObj(
  "latitude" -> JsNum(54.409472),
  "longitude" -> JsNum(18.584570)
)
```

Type class derivation - motivation

```
case class Company(name: String,
                   address: Address,
                   location: Location)

implicit val companyInstance: JsonEncoder[Company] =
  new JsonEncoder[Company] {
    type Out = JsObj
    def toJson(com: Company): Out = JsObj(
      "name" -> com.name.toJson,
      "address" -> com.address.toJson,
      "location" -> com.location.toJson
    )
  }
```

Type class derivation - motivation

```
val scalacAddr = Address("Poland", "Gdańsk", "Bora-Komorowskiego 19B", "80-377")
val scalacLocation = Location(54.409472, 18.584570)
val scalac = Company("Scalac", scalacAddr, scalacLocation)

scalac.toJson == JsObj(
  "name" -> "Scalac",
  "address" -> JsObj(
    "country" -> JsStr("Poland"),
    "city" -> JsStr("Gdańsk"),
    "street" -> JsStr("Bora-Komorowskiego 19B"),
    "postalCode" -> JsStr("80-377")
  ),
  "location" -> JsObj(
    "latitude" -> JsNum(54.409472),
    "longitude" -> JsNum(18.584570)
  )
)
```

Type class derivation - motivation

- we need to define implicit instance for every new type that we introduced
- with every implicit instance we have to repeat ourselves (few times)
- … and have a chance for making a stupid mistake
- defining type class instances manually means lot of boilerplate code

Type class derivation - idea

- our instances for case classes were very similar to each other
- generate the code for user-defined case classes
- eliminate boilerplate code, increase safety and user experience

Type class derivation - implementation

There are few possible ways to implement type class derivation in Scala:

- runtime reflection
- compile-time reflection (macros)
- type-level metaprogramming

Type class derivation with Shapeless

```scala
case class Location(latitude: Double, longitude: Double)
```

↓

```scala
Double :: Double :: HNil
```

↓

```scala
JsonEncoder[Double :: Double :: HNil]
```

↓

```scala
def toJson(obj: Double :: Double :: HNil): JsValue
```

Type class derivation with Shapeless

```
case class Location(latitude: Double, longitude: Double)
```

```
Double :: Double :: HNil
```

```
JsonEncoder[Double :: Double :: HNil]
```

```
def toJson(obj: Double :: Double :: HNil): JsArr
```

Type class derivation - ex4 (part 1)

```
val scalacLocation = Location(54.409472, 18.584570)

scalacLocation.toJson == JsArr(JsNum(54.409472), JsNum(18.584570))
```

Type class derivation - `ex4 (part 1)`

- go to `scalawave.exercise.ex4.part1` package
- run `sbt "~testOnly scalawave.exercise.ex4.part1.JsonEncoderSpec"`
- your solution is ready when all your tests pass

Missing field labels

- JSON arrays is not what we want
- we would want to encode case classes as JSON objects
- we need to carry typelevel information about field lables

What is a type?

What is a type?

Type = set of values

What is a type?

Type = set of values

```
Boolean = {true, false}
Int = {Int.minValue, …, -1, 0, 1, …, Int.maxValue}
Char = {'\u0000', '\u0000', …, '\uffff'}
```

What is a type?

Type = set of values

```
Boolean = {true, false}
Int = {Int.minValue, …, -1, 0, 1, …, Int.maxValue}
Char = {'\u0000', '\u0000', …, '\uffff'}
String = Char*
```

What is a type?

Type = set of values

```
Boolean = {true, false}
Int = {Int.minValue, …, -1, 0, 1, …, Int.maxValue}
Char = {'\u0000', '\u0000', …, '\uffff'}
String = Char*
List[A] = A*
List[Boolean] = {Nil, List(true), List(false),
  List(true, true), List(true, false), …}
```

What is a type?

Type = set of values

```
Unit = {()}
Null = {null}
Nothing = {}
Any = sum set of all other types
```

What is a type?

Type = set of values

```
(A, B) = A × B
(A, B, C) = A × B × C
Either[A, B] = A ∪ B
```

What is a type?

Type = set of values

```
(A, B) = A × B
(A, B, C) = A × B × C
Either[A, B] = A ∪ B

(Boolean, Unit) = Boolean × Unit
                = {(true, ()), (false, ())}
```

What is a type?

Relation between types

```
Nothing ⊆ String ⊆ Any
Nothing ⊆ Int ⊆ Any
∅ ⊆ {true, false} ⊆ Any
```

What is a type?

Relation between types

```
Nothing ⊆ String ⊆ Any
Nothing ⊆ Int ⊆ Any
∅ ⊆ {true, false} ⊆ Any

Nothing <: String <: Any
Nothing <: Int <: Any
Nothing <: Boolean <: Any
```

What is a type?

Relation between values/types

```
"hello" ∈ String
5 ∈ Int
false ∈ Boolean
```

What is a type?

Relation between values/types

```
"hello" ∈ String
      5 ∈ Int
  false ∈ Boolean

"hello" : String
      5 : Int
  false : Boolean
```

What is a type?

More specific ~~sets~~ types

```
PositiveInt = {1, 2, …, Int.maxValue}
    EvenInt = {…, -2, 0, 2, 4, …}
FibonacciInt = {1, 2, 3, 5, 8, 13, …}

PositiveInt ⊆ Int
EvenInt ⊆ Int
FibonacciInt ⊆ Int
```

What is a type?

More specific ~~sets~~ types

```
Palindrome = {w ∈ String | w == w.reverse}
CountryCodes = {"PL", "UK", "DE", "US", …}
```

```
Palindrome ⊆ String
CountryCodes ⊆ String
```

Singleton types

Even more specific ~~sets~~ types

```
Int(5) = {5}
Int(42) = {42}
Char('x') = {'x'}
String("abc") = {"abc"}

Int(5) ⊆ Int
Int(42) ⊆ Int
Char('x') ⊆ Char
String("abc") ⊆ String
```

Singleton types

Even more specific ~~sets~~ types

```
5 ∈ Int(5)
42 ∈ Int(42)
'x' ∈ Char('x')
"abc" ∈ String("abc")

5 : Int(5)
42 : Int(42)
'x' : Char('x')
"abc" : String("abc")
```

Singleton types

Singleton type is a type that has only one possible value

Working with singleton types

- Scala has rather limited support for singleton types
- We can’t express arbitrary singleton types
- SIP-23 - Literal based singleton types
- will be better since 2.11.9 / 2.12.1

Working with singleton types

```
Welcome to Scala 2.12.0

scala> val x = "hello"
x: String = hello

scala> val y: x.type = x
y: x.type = hello
```

Working with singleton types

```
Welcome to Scala 2.12.0

scala> val x = "hello"
x: String = hello

scala> val y: x.type = "foo"
<console>:12: error: type mismatch;
 found   : String("foo")
 required: x.type
       val y: x.type = "foo"
```

Working with singleton types

```
Welcome to Scala 2.12.0

scala> val x = "hello"
x: String = hello

scala> val y: x.type = "hello"
<console>:12: error: type mismatch;
 found   : String("hello")
 required: x.type
       val y: x.type = "hello"
```

SIP-23

```
scala> val x: "hello".type  = "hello"
x: "hello".type = hello

scala> val y: 42.type = 42
y: 42.type = 42

scala> type N = 10.type
defined type alias N
```

SIP-23

```
scala> val x: "hello" = "hello"
x: "hello" = hello

scala> val y: 42 = 42
y: 42 = 42

scala> type N = 10
defined type alias N
```

Shapeless support for singleton types

- `.narrow`
- `Witness.Aux`
- `FieldType`
- record types
- `LabelledGeneric`

Shapeless support for singleton types

```
scala> import shapeless.syntax.singleton._
import shapeless.syntax.singleton._

scala> val x = "abc".narrow
x: abc = "abc"
scala> val y = 'abc.narrow
y: Symbol with tag.Tagged[abc] = 'abc
scala> val z = 42.narrow
z: 42 = 42
```

Singleton type materialization

```
def valueOf[T]: T = ???
```

Witness typeclass

```
trait Witness {
  type T
  val value: T {}
}

object Witness {
  type Aux[T0] = Witness { type T = T0 }

  implicit def apply[T]: Witness.Aux[T] = macro …
}
```

Witness typeclass

```
scala> Witness[x.type]
res4: Witness.Aux[x.type] = shapeless.Witness$$anon$1@2a28d858

scala> Witness[x.type].value
res5: x.type = "abc"

scala> implicitly[Witness.Aux[x.type]]
res6: Witness.Aux[x.type] = shapeless.Witness$$anon$1@5c7808bc

scala> implicitly[Witness.Aux[x.type]].value
res7: x.type = "abc"
```

Singleton type materialization

```
def valueOf[T](implicit w: Witness.Aux[T]): T = w.value
```

```
scala> val a = "hello".narrow
a: hello = "hello"
scala> val aValue = valueOf[a.type]
aValue: a.type = "hello"
```

Missing field labels

- JSON arrays is not what we want
- we would want to encode case classes as JSON objects
- <u>we need to carry typelevel information about field lables</u>

use singleton types

Shapeless field types

```
object labelled {

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

- `K` is type-level, compile-time annotation
- `FieldType[K, V] <: V`
- just because for any `A`, `B`:
  `A with B <: A` and `A with B <: B`

Shapeless field types

```scala
object labelled {

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

```
scala> import shapeless.syntax.singleton._
import shapeless.syntax.singleton._

scala> "the answer" ->> 42
res1: Int with labelled.KeyTag[the answer, Int] = 42

scala> 'ans ->> 42.0
res2: Double with labelled.KeyTag[Symbol with tag.Tagged[ans], Double] = 42.0
```

Shapeless field types

```
scala> ("the answer" ->> 42) + 5
res2: Int = 47
```

Shapeless field types

```
def extractKey[K, V](field: FieldType[K, V])
                    (implicit k: Witness.Aux[K]): K =
  k.value
```

```
scala> extractKey("the answer" ->> 42)
res1: String = the answer
```

Shapeless records

```scala
case class Location(latitude: Double, longitude: Double)
```

```scala
Double :: Double :: HNil
```

```scala
FieldType[latitude, Double] :: FieldType[longitude, Double] :: HNil
```

Shapeless records

```
scala> "latitude" ->> 52.13 :: "longitude" ->> 18.22 :: HNil
res1: Double with labelled.KeyTag[latitude, Double] :: Double with
labelled.KeyTag[longitude, Double] :: HNil = 52.13 :: 18.22 :: HNil
```

Shapeless LabelledGeneric

```scala
case class Car(brand: String,
               model: String,
               productionYear: Int)
```

to: Car => Repr

from: Repr => Car

```scala
type Repr = String with KeyTag[Symbol with Tagged[String("brand")],String]
  :: String with KeyTag[Symbol with Tagged[String("model")],String]
  :: Int with KeyTag[Symbol with Tagged[String("productionYear")],Int]
  :: HNil
```

Shapeless LabelledGeneric

```
val carLabGen = implicitly[LabelledGeneric[Car]]
// LabelledGeneric[Car] { type Repr = … }

val myCar = Car("Cadillac", "DeVille", 1961)

val myCarRecord = carLabGen.to(c)
// Cadillac :: DeVille :: 1961 :: HNil  :  carLabGen.Repr

val myCar2 = carLabGen.from(myCarRecord)
// Car(Cadillac, DeVille, 1961) : Car

val maluch = carLabGen.from('brand ->> "Fiat" ::
   'model ->> "126P" :: 'productionYear ->> 1976 :: HNil)
// Car(Fiat, 126P, 1976) : Car
```

Shapeless LabelledGeneric

```
val myCarRecord = carLabGen.to(c)
// Cadillac :: DeVille :: 1961 :: HNil  :  carLabGen.Repr

extractKey(myCarRecord.head)
// 'brand : Symbol with tag.Tagged[brand]

extractKey(myCarRecord.head).name
// "brand" : String
```

Type class derivation - `ex4 (part 2)`

- go to `scalawave.exercise.ex4.part2` package
- run `sbt "~testOnly scalawave.exercise.ex4.part2.JsonEncoderSpec"`
- your solution is ready when all your tests pass

Type class derivation - current status

Now our JSON type class derivation should work for:

- primitive types
- product types (case classes and tuples)

We would want it to work for coproduct types as well, meaning:

- `Either[A, B]`
- `sealed` hierarchies

Coproducts

```
Either[String, Int]
```

```
Left("abc")
```

```
Right(42)
```

```
sealed trait ServiceResponse
```

```
ValidationError("xyz")
```

```
DBReadError("abc")
```

```
Result(42)
```

Coproducts

```
sealed abstract class Either[+A, +B]
case class Left[+A, +B](value: A) extends Either[A, B]
case class Right[+A, +B](value: B) extends Either[A, B]

sealed abstract class Try[+T]
case class Failure[+T](exception: Throwable) extends Try[T]
case class Success[+T](value: T) extends Try[T]
```

Coproducts: sealed hierarchies

Just like tuples/case classes, sealed hierarchies have some problems:

- don’t have recursive structure
- doesn’t need to carry type-level information
- hard to define type-level computation with

Shapeless Coproduct

```
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 {
  def impossible: Nothing
}
```

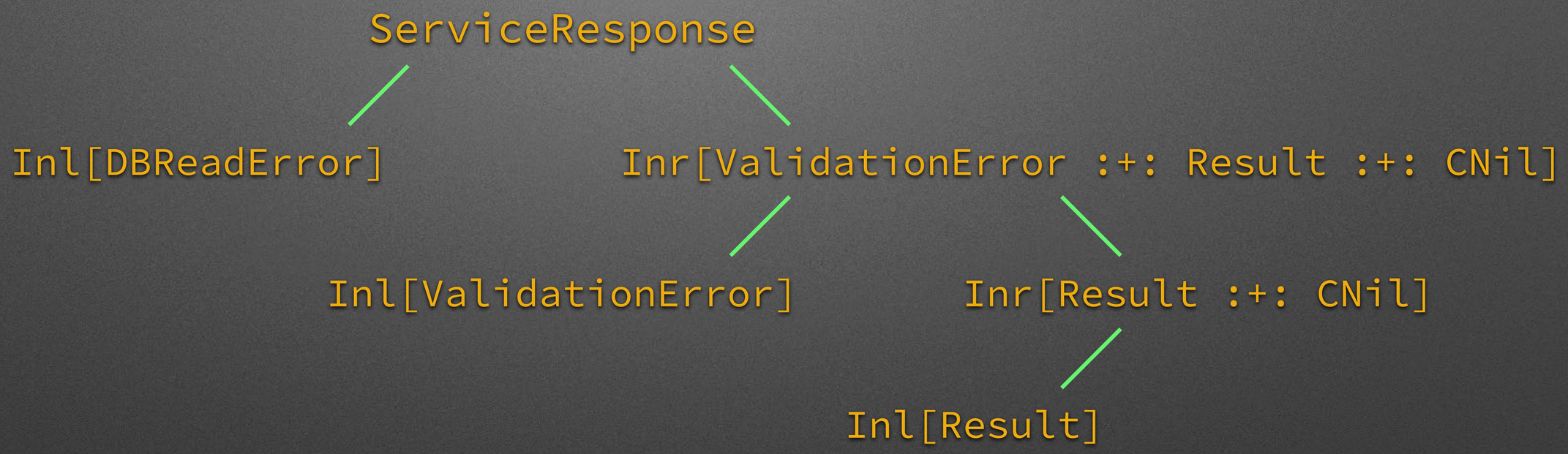
Shapeless Coproduct

```
case class DBReadError(reason: String)
case class ValidationError(error: String)
case class Result(value: Int)

type ServiceResponse = DBReadError :+: ValidationError :+: Result :+: CNil

Inl(DBReadError("why")) : ServiceResponse
Inr(Inl(ValidationError("why"))) : ServiceResponse
Inr(Inr(Inl(Result(42)))) : ServiceResponse
```

Shapeless Coproduct




```
type ServiceResponse = DBReadError :+: ValidationError :+: Result :+: CNil

Inl(DBReadError("why")) : ServiceResponse
Inr(Inl(ValidationError("why"))) : ServiceResponse
Inr(Inr(Inl(Result(42)))) : ServiceResponse
```

Shapeless Coproduct

```
type ServiceResponse = DBReadError :+: ValidationError :+: Result :+: CNil

Inl(DBReadError("why")) : ServiceResponse
Inr(Inl(ValidationError("why"))) : ServiceResponse
Inr(Inr(Inl(Result(42)))) : ServiceResponse

type ServiceResponse2 = Either[DBReadError, Either[ValidationError, Result]]

Left(DBReadError("why")) : ServiceResponse2
Right(Left(ValidationError("why"))) : ServiceResponse2
Right(Right(Result(42))) : ServiceResponse2
```

Generic for coproducts

```
sealed trait ServiceResponse
case class DBReadError(reason: String) extends ServiceResponse
case class ValidationError(error: String) extends ServiceResponse
case class Result(value: Int) extends ServiceResponse

val srGen = Generic[ServiceResponse]
srGen: Generic[ServiceResponse] {
  type Repr = DBReadError :+: Result :+: ValidationError :+: CNil
}

srGen.to(Result(42))
res1: srGen.Repr = Inr(Inl(Result(42)))
```

JSON encoding for coproducts

```
val response1: ServiceResponse = DBReadError("why")
response1.toJson = JsObj("reason" -> JsStr("why"))
```

```
val response2: ServiceResponse = ValidationError("why")
response2.toJson = JsObj("error" -> JsStr(„why"))
```

JSON encoding for coproducts

```
val response1: ServiceResponse = DBReadError("why")
response1.toJson = JsObj("DBReadError" ->
  JsObj("reason" -> JsStr("why"))
)

val response2: ServiceResponse = ValidationError("why")
response2.toJson = JsObj("ValidationError" ->
  JsObj("error" -> JsStr("why"))
)
```

LabelledGeneric for coproducts

```
sealed trait ServiceResponse
case class DBReadError(reason: String) extends ServiceResponse
case class ValidationError(error: String) extends ServiceResponse
case class Result(value: Int) extends ServiceResponse

val srLabGen = LabelledGeneric[ServiceResponse]
srLabGen: LabelledGeneric[ServiceResponse] {
  type Repr =
      FieldType[Symbol with Tag["DBReadError"], DBReadError]
  :+: FieldType[Symbol with Tag["Result"], Result]
  :+: FieldType[Symbol with Tag["ValidationError"], ValidationError]
  :+: CNil
}
```

Type class derivation - `ex4 (part 3)`

- go to `scalawave.exercise.ex4.part3` package
- run `sbt "~testOnly scalawave.exercise.ex4.part3.JsonEncoderSpec"`
- your solution is ready when all your tests pass

Type class derivation - current status

Now our JSON type class derivation should work for:

- primitive types
- product types (case classes and tuples)
- coproduct types (`Either`, `Try`, `sealed` hierarchies)

Type class derivation - current status

But that's not all, still we have some things to improve

- missing dedicated encoding for popular scala types like `Option`, `List`, `Map`
- missing support for nested data structures
- missing support for recursive data structures

Current support for `Option`

```
val someInt: Option[Int] = Some(5)
someInt.toJson
// JsObj("Some" -> JsObj("value" -> JsNum(5)))
someInt.toJsonString
// {"Some": {"value": 5}}

val noneInt: Option[Int] = None
noneInt.toJson
// JsObj("None" -> JsObj())
noneInt.toJsonString
// {"None": {}}
```

Dedicated support for `Option`

```
val someInt: Option[Int] = Some(5)
someInt.toJson
// JsNum(5)
someInt.toJsonString
// 5

val noneInt: Option[Int] = None
noneInt.toJson
// JsNull
noneInt.toJsonString
// null
```

Dedicated support for `Option`

```
implicit def optionEnc[T]: JsonEncoder.Aux[Option[T], JsValue] = ???
```

Dedicated support for `List, Map`

```
implicit def listEnc[T]: JsonEncoder.Aux[List[T], JsArr] = ???

implicit def mapEnc[K, V]: JsonEncoder.Aux[Map[K, V], JsObj] = ???
```

Lack of support for nested products

```
scala> (2, "abc").toJson
res1: JsObj = JsObj(List((_1,JsNum(2)), (_2,JsStr(abc))))

scala> (2, ("abc", true)).toJson
<console>:18: error: diverging implicit expansion for type
JsonEncoder.Aux[(Int, (String, Boolean)),O]
starting with method genEnc in object JsonEncoder
       (2, ("abc", true)).toJson
```

Lack of support for nested products

```scala
case class Author(firstName: String, lastName: String)
case class Book(title: String, author: Author)
```

```
scala> val cormen = Book("Introduction to Algorithms",
Author("Thomas", "Cormen"))
cormen: Book = Book(Introduction to Algorithms,Author(Thomas,Cormen))

scala> cormen.toJson
<console>:19: error: diverging implicit expansion for type
JsonEncoder.Aux[Book,O]
starting with method genEnc in object JsonEncoder
       cormen.toJson
```

Lack of support for recursive structures

```scala
sealed trait MyList[+A]
case object MyNil extends MyList[Nothing]
case class MyCons[+T](head: T, tail: MyList[T]) extends MyList[T]
```

```
scala> val myListInstance: MyList[Int] = MyCons(1, MyCons(2, MyNil))
myListInstance: MyList[Int] = MyCons(1,MyCons(2,MyNil))

scala> myListInstance.toJson
<console>:19: error: diverging implicit expansion for type
JsonEncoder.Aux[MyList[Int],O]
starting with method genEnc in object JsonEncoder
       myListInstance.toJson
```

What’s wrong with generic instance?

```
implicit def genEnc[T, Repr, O <: JsValue]
                (implicit gen: LabelledGeneric.Aux[T, Repr],
                          enc: JsonEncoder.Aux[Repr, O])
  : JsonEncoder.Aux[T, O] = …
```

Divergent implicits

```
JsonEncoder.Aux[Book, _]
JsonEncoder.Aux[String :: Author :: HNil, _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[Author, _]
JsonEncoder.Aux[String :: String :: HNil, _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[HNil, _]
JsonEncoder.Aux[HNil, _]
```

Divergent implicits

```
JsonEncoder.Aux[Book, _]
JsonEncoder.Aux[String :: Author :: HNil, _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[Author, _]
JsonEncoder.Aux[String :: String :: HNil, _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[HNil, _]
JsonEncoder.Aux[HNil, _]
```

Divergent implicits

```
JsonEncoder.Aux[Book, _]
JsonEncoder.Aux[::[String, ::[Author, HNil]], _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[Author, _]
JsonEncoder.Aux[::[String, ::[String, HNil]], _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[HNil, _]
JsonEncoder.Aux[HNil, _]
```

Divergent implicits

```
JsonEncoder.Aux[Book, _]
JsonEncoder.Aux[::[_, ::[_, _]], _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[Author, _]
JsonEncoder.Aux[::[_, ::[_, _]], _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[HNil, _]
JsonEncoder.Aux[HNil, _]
```

Divergent implicits

```
JsonEncoder.Aux[Book, _]
JsonEncoder.Aux[::[_, ::[_, _]], _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[Author, _]
JsonEncoder.Aux[::[_, ::[_, _]], _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[HNil, _]
JsonEncoder.Aux[HNil, _]
```

I've seen it before!

Divergent implicits

```
JsonEncoder.Aux[Book, _]
JsonEncoder.Aux[::[_, ::[_, _]], _]
JsonEncoder.Aux[String, _]
JsonEncoder.Aux[Author, _]
JsonEncoder.Aux[::[_, ::[_, _]], _]
error: divergent implicit expansion
```

I've seen it before!

Shapeless Lazy

```
trait Lazy[+T] {
  val value: T
}

object Lazy {
  implicit def mkLazy[I]: Lazy[I] =
    macro LazyMacrosRef.mkLazyImpl[I]
}
```

Using Shapeless Lazy

```
implicit def hconsEnc[K <: Symbol, V, T <: HList]
                (implicit key: Witness.Aux[K],
                          headEnc: JsonEncoder[V],
                          tailEnc: JsonEncoder.Aux[T, JsObj])
    : JsonEncoder.Aux[FieldType[K, V] :: T, JsObj] = …

implicit def coproductEnc[K <: Symbol, V, T <: Coproduct]
                (implicit key: Witness.Aux[K],
                          headEnc: JsonEncoder[V],
                          tailEnc: JsonEncoder.Aux[T, JsObj])
    : JsonEncoder.Aux[FieldType[K, V] :+: T, JsObj] = …

implicit def genEnc[T, Repr, O <: JsValue]
                (implicit gen: LabelledGeneric.Aux[T, Repr],
                          enc: JsonEncoder.Aux[Repr, O])
  : JsonEncoder.Aux[T, O] = …
```

Using Shapeless Lazy

```
implicit def hconsEnc[K <: Symbol, V, T <: HList]
                (implicit key: Witness.Aux[K],
                          headEnc: Lazy[JsonEncoder[V]],
                          tailEnc: JsonEncoder.Aux[T, JsObj])
    : JsonEncoder.Aux[FieldType[K, V] :: T, JsObj] = …

implicit def coproductEnc[K <: Symbol, V, T <: Coproduct]
                (implicit key: Witness.Aux[K],
                          headEnc: Lazy[JsonEncoder[V]],
                          tailEnc: JsonEncoder.Aux[T, JsObj])
    : JsonEncoder.Aux[FieldType[K, V] :+: T, JsObj] = …

implicit def genEnc[T, Repr, O <: JsValue]
                (implicit gen: LabelledGeneric.Aux[T, Repr],
                          enc: Lazy[JsonEncoder.Aux[Repr, O]])
  : JsonEncoder.Aux[T, O] = …
```

Type class derivation - `ex4 (part 4)`

- go to `scalawave.exercise.ex4.part4` package
- run `sbt "~testOnly scalawave.exercise.ex4.part4.JsonEncoderSpec"`
- your solution is ready when all your tests pass

Type class derivation - summary

We were able to define JSON type class derivation that works for:

- primitive types
- product types (case classes and tuples)
- coproduct types (`Either`, `Try`, `sealed` hierarchies)
- nested product types
- recursive types
- `Option`, `List`, `Map` with special, optimized encoding

Type class derivation: summary

- type safety, but need to introduce some abstract concepts
- run time overhead of converting to generic representations
- poor missing instance reporting

Metaprogramming alternatives

Implicit macros

- better control of error reporting
- less runtime overhead (no intermediate representation)
- low-level compiler API

Type-level programming - examples

Type class derivation

real life examples

- serializations, conversion between formats (circe)
- random instance generators (scalacheck-shapeless)
- automatic database mappings
- type-safe query DSL (constraining SQL at type level)

Shapeless as standard type-level library

Shapeless as standard library

- shapeless data types (`Nat`, `HList`, `Coproduct`)
- shapeless macros (`Generic`, `LabelledGeneric`, `Witness`)
- shapeless type-level computations (`shapeless.ops._`)
- polymorphic functions

Type-level programming - summary

Learning materials

- The Type Astronaut's Guide to Shapeless
  https://github.com/underscoreio/shapeless-guide
- Typelevel Blog
  http://typelevel.org/blog/
- Shapeless: not a tutorial
  http://kanaka.io/blog/2015/11/09/shapeless-not-a-tutorial-part-1.html

Learning materials

- Scala Exercises: Shapeless
  https://www.scala-exercises.org/shapeless
- Miles Sabin @ Dublin Scala Users Group
  https://www.youtube.com/watch?v=LaOiBdrM8uo
- Roll Your Own Shapeless - by Daniel Spiewak
  https://www.youtube.com/watch?v=GKIfu1WtSz4

The end

That’s all, thank you!