



Hi! 🙌

My name is  @folone

Type level programming in Scala¹



¹ with lies

```
sealed trait Bool {  
    type &&[B <: Bool] <: Bool  
    type ||[B <: Bool] <: Bool  
    type IfElse[T, F] <: Any  
}  
  
trait True extends Bool {  
    type &&[B <: Bool] = B  
    type ||[B <: Bool] = True  
    type IfElse[T, F] = T  
}  
  
trait False extends Bool {  
    type &&[B <: Bool] = False  
    type ||[B <: Bool] = B  
    type IfElse[T, F] = F  
}
```


<https://github.com/folone/type-level-birds>



There's a Smalltalk in your Scala

– Stefan Zeiger - Type level Computations in Scala -
ScalalO - 2015

A black and white photograph of a large group of people, likely employees, standing in rows in a factory or industrial setting. They are dressed in casual clothing, some with visible logos like "TRACE" and "VOLKSWAGEN". The background shows the complex steel framework and pipes of the factory ceiling.

Peano numbers!


```
trait MinusOne[A <: Nat] {
    type Res <: Nat
}

object MinusOne {
    implicit val baseCase: MinusOne[Z] { type Res = Z } =
        new MinusOne[Z] {
            type Res = Z
        }
    implicit def inductiveCase[A <: Nat]: MinusOne[Succ[A]] { type Res = A } =
        new MinusOne[Succ[A]] {
            type Res = A
        }
}
```



```
trait Plus[A <: Nat, B <: Nat] {
    type Res <: Nat
}
object Plus {
    implicit def baseCase[A <: Nat]: Plus[A, Z] { type Res = A } =
        new Plus[A, Z] {
            type Res = A
        }
    implicit def inductiveCase[A <: Nat, B <: Nat, C <: Nat, D <: Nat]
        (implicit ev0: MinusOne[B] { type Res = C },
         ev1: Plus[Succ[A], C] { type Res = D }): Plus[A, B] { type Res = D } =
        new Plus[A, B] {
            type Res = D
        }
}
```

```
implicit def inductiveCase[A <: Nat,  
                         B <: Nat,  
                         C <: Nat,  
                         D <: Nat]  
(implicit ev0: MinusOne[B] { type Res = C },  
     ev1: Plus[Succ[A], C] { type Res = D }): Plus[A, B] { type Res = D } =  
new Plus[A, B] {  
    type Res = D  
}
```

```
trait Plus[A <: Nat, B <: Nat] {
    type Res <: Nat
}
object Plus {
    type Aux[A <: Nat, B <: Nat, Res1 <: Nat] = Plus[A, B] { type Res = Res1 }

    implicit def baseCase[A <: Nat]: Aux[A, Z, A] = new Plus[A, Z] {
        type Res = A
    }
    implicit def inductiveCase[A <: Nat, B <: Nat, C <: Nat, D <: Nat]
        (implicit ev0: MinusOne.Aux[B, C],
         ev1: Plus.Aux[Succ[A], C, D]): Aux[A, B, D] =
        new Plus[A, B] {
            type Res = D
        }
}
```

```
type _1 = Succ[Z]
type _2 = Succ[_1]
type _3 = Succ[_2]
```

```
@ implicitly[Plus[_1, _2]]
res0: Plus[_1, _2] { type Res = _3 } = Plus$$anon$2@f79a760
```

```
@ implicitly[Plus[_1, _2]]  
res0: Plus[_1, _2] = Plus$$anon$2@13c3c1e1
```

```
@ implicitly[Plus[_1, _2]#Res =:= _3]  
res1: =:=[Plus[_1, _2]#Res ,_3] = <function1>
```

```
@ implicitly[Plus.Aux[_1, _2, _3]]  
res0: Plus.Aux[_1, _2, _3] = Plus.Aux$$anon$2@3d08f3f5
```

```
@ implicitly[Plus.Aux[_1, _2, Z]]  
<console>:15: error: could not find implicit value for parameter e: Plus.Aux[_1, _2, Z]  
implicitly[Plus.Aux[_1, _2, Z]]  
^
```

Filists




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

object Length {
    implicit val baseCase: Length[HNil.type] { type Res = Z } =
        new Length[HNil.type] {
            type Res = Z
        }
    implicit def inductiveCase[H,
                                T <: HList,
                                N <: Nat]
        (implicit ev0: Length[T] { type Res = N }) =
        new Length[HCons[H, T]] {
            type Res = Succ[N]
        }
}
```

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

object Length {
    type Aux[L <: HList, Res1 <: Nat] = Length[L] { type Res = Res1 }
    implicit val baseCase: Aux[HNil.type, Z] = new Length[HNil.type] {
        type Res = Z
    }
    implicit def inductiveCase[H, T <: HList, N <: Nat]
        (implicit ev0: Length.Aux[T, N]) =
        new Length[HCons[H, T]] {
            type Res = Succ[N]
        }
}
```

shapeless!

<https://github.com/milessabin/shapeless>

```
@ import shapeless._  
import shapeless._  
  
@ val hlist = 11 :: "hello" :: HNil  
hlist: Long :: String :: HNil = 1 :: hello :: HNil
```

```
@ hlist(0)
res7: Long = 1
```

```
@ hlist(1)
res8: String = hello
```

```
@ hlist(2)
<console>:16: error:
Implicit not found: Scary[Type].Please#Ignore
You requested to access an element at the position
TypelevelEncodingFor[2.type]
but the HList Long :: String :: HNil is too short.
```

```
    hlist(2)
        ^

```

```
Compilation failed.
```

There's a Prolog in your Scala

-- ScalaIO 2015

Json serialization



```
def json(o: Any): String
```

```
def json(o: Any): JsonAst
```

```
def json[A : Writes](o: A): JsonAst

trait Writes[A] {
  def write(o: A): JsonAst
}
```

```
@ import play.api.libs.json.{Json => PJson}  
import play.api.libs.json.{Json => PJson}
```

```
@ case class Thing(id: Long, payload: String)  
defined class Thing
```

```
@ PJson.writes[Thing]  
res0: Writes[Thing] = Writes$$anon$2@f79a760
```

```
@ case class Omg(_1: Int, _2: Int, _3: Int, _4: Int, _5: Int,  
_6: Int, _7: Int, _8: Int, _9: Int, _10: Int, _11: Int, _12: Int,  
_13: Int, _14: Int, _15: Int, _16: Int, _17: Int, _18: Int,  
_19: Int, _20: Int, _21: Int, _22: Int, _23: Int)  
defined class Omg  
@ PJson.writes[Omg]  
cmd9.sc:1: No unapply or unapplySeq function found for class Omg.  
val res9 = PJson.writes[Omg]  
^
```

Compilation Failed



>700LOC of macro



Thing(id: Long, payload: String)

(Long, String)

Long :: String :: Nil

1. Case classes are essentially tuples (or HLists) with names.
Can we transform one to another?
2. Define `Writes` for an `HList`:
 1. How do we serialize an `HNil`?
 2. How do we serialize an `HCons`?
3. 

Two (mandatory) building blocks

- HLists
- Generic²

² lie: what we actually need is a LabelledGeneric

HList

```
@ import shapeless._  
import shapeless._  
  
@ val hlist = 11 :: "hello" :: HNil  
hlist: Long :: String :: HNil = 1 :: hello :: HNil
```

```
@ hlist(0)
res7: Long = 1
```

```
@ hlist(1)
res8: String = hello
```

```
@ hlist(2)
<console>:16: error:
Implicit not found: Scary[Type].Please#Ignore
You requested to access an element at the position
TypelevelEncodingFor[2.type]
but the HList Long :: String :: HNil is too short.
```

```
    hlist(2)
        ^

```

```
Compilation failed.
```

Generic

```
@ case class Thing(id: Long, payload: String)  
defined class Thing
```

```
@ val generic = Generic[Thing]  
generic: shapeless.Generic[Thing]{type Repr = Long :: String :: HNil} =  
anon$macro$3$1@7f8f5e52
```

```
@ val representation = generic.to(Thing(1, "hello"))
representation: res0.Repr = 1 :: hello :: HNil
```

```
@ representation(0)
res10: Long = 1
```

```
@ representation(1)
res11: String = hello
```

```
@ representation(2)
<console>:19: error:
Implicit not found: Scary[Type].Please#Ignore
You requested to access an element at the position
TypelevelEncodingFor[2.type]
but the HList Long :: String :: HNil is too short.
representation(2)
^
```

```
@ generic.from(hlist)
res7: Thing = Thing(1L, "hello")
```

Putting this together

1. Let shapeless transform our case classes into some unified form (HLists)
2. Write down instances of the Writes typeclass for those forms in a generic way (HNil, HCons)
3. Let the magic flow 

```
object writes extends LabelledProductTypeClassCompanion[Writes] with DefaultWrites {
  object typeClass extends LabelledProductTypeClass[Writes] {
    override def emptyProduct: Writes[HNil] =
      Writes(_ => PlayJson.obj())

    override def product[H, T <: HList](name: String, headEv: Writes[H], tailEv: Writes[T]) =
      Writes[H :: T] {
      case head :: tail =>
        val h = headEv.writes(head)
        val t = tailEv.writes(tail)
        PlayJson.obj(name -> h) ++ t
    }

    override def project[F, G](instance: => Writes[G], to: F => G, from: G => F) =
      Writes[F](f => instance.writes(to(f)))
  }
}
```



```
object writes extends LabelledProductTypeClassCompanion[Writes] with DefaultWrites {
  object typeClass extends LabelledProductTypeClass[Writes] {
    override def emptyProduct: Writes[HNil] =
      Writes(_ => PlayJson.obj())

    override def product[H, T <: HList](name: String, headEv: Writes[H], tailEv: Writes[T]) =
      Writes[H :: T] {
      case head :: tail =>
        val h = headEv.writes(head)
        val t = tailEv.writes(tail)
        PlayJson.obj(name -> h) ++ t
    }

    override def project[F, G](instance: => Writes[G], to: F => G, from: G => F) =
      Writes[F](f => instance.writes(to(f)))
  }
}
```

```
object writes extends LabelledProductTypeClassCompanion[Writes] with DefaultWrites {
  object typeClass extends LabelledProductTypeClass[Writes] {
    override def emptyProduct: Writes[HNil] =
      Writes(_ => PlayJson.obj())

    override def product[H, T <: HList](name: String, headEv: Writes[H], tailEv: Writes[T]) =
      Writes[H :: T] {
      case head :: tail =>
        val h = headEv.writes(head)
        val t = tailEv.writes(tail)
        PlayJson.obj(name -> h) ++ t
    }

    override def project[F, G](instance: => Writes[G], to: F => G, from: G => F) =
      Writes[F](f => instance.writes(to(f)))
  }
}
```

```
object writes extends LabelledProductTypeClassCompanion[Writes] with DefaultWrites {
  object typeClass extends LabelledProductTypeClass[Writes] {
    override def emptyProduct: Writes[HNil] =
      Writes(_ => PlayJson.obj())

    override def product[H, T <: HList](name: String, headEv: Writes[H], tailEv: Writes[T]) =
      Writes[H :: T] {
      case head :: tail =>
        val h = headEv.writes(head)
        val t = tailEv.writes(tail)
        PlayJson.obj(name -> h) ++ t
    }

    override def project[F, G](instance: => Writes[G], to: F => G, from: G => F) =
      Writes[F](f => instance.writes(to(f)))
  }
}
```



The Type Astronaut's Guide to Shapeless

Dave Gurnell
foreword by Miles Sabin

By @davegurnell



@folone