Introduction to:

Recursive implicit proofs with Shapeless HLists

Vladimir Pavkin

Riga Scala Meetup

18.08.2016

Agenda

1. Intro: another look at implicits
2. Simple recursive proof with `scala.math.Ordering`
3. Math proofs vs. Implicit-based proofs
4. `HList` basics
5. "Pure" proof for `HList`: `OddProduct`
6. `Show` typeclass and it's generic proof
7. Expanding `HList` proof to all product-like types with `shapeless.Generic`
8. Proofs with type-level computations: generic product diffs.
   (given we have time)

CODE SAMPLES

https://github.com/vpavkin/recursive-implicit-proofs-talk

Another look at implicits

Writing a **program** for a *task*

is very similar to

constructing a **proof** for a *theorem*.

Compiler

+

Type System

=

Proof construction tool

Typeclass – a property that can be proved

```
Ordering[A]
```

Implicit typeclass instance – a proved fact that some type has particular property

```
implicit val intOrdering =
new Ordering[Int] {…}
```

Implicit method parameter – a requirement that is proved by the compiler at every call site.

Implicit scope – the proof system. A set of axioms that is used by compiler to build more complex proofs.

Implicit not found – proof can't be constructed.

```
def sortBy[B](f: A => B)
(implicit ord: Ordering[B])

//or

def sortBy[B: Ordering](f: A => B)
```

```
implicit val intOrdering = …

implicit def optOrdering[T:Ordering] =
new Ordering[Option[T]] { … }

// Ok
implicitly[Ordering[Option[Int]]]

// Can't prove
implicitly[Ordering[Option[String]]]
```

Simple recursive proof with `scala.math.Ordering`

(Demo)

Math proofs vs Scala implicit proofs

Math proofs vs Scala implicit proofs

A

```
// type A = Int

implicit val intOrdering: Ordering[Int] = …
```

Math proofs vs Scala implicit proofs

A => B

```
// type B = Option[A]

implicit def optOrdering[A]
(implicit ev: Ordering[A]): Ordering[Option[A]] = …
```

A & B => C

```
// type C = (A, B)

implicit def tuple2Ordering[A, B]
(implicit A: Ordering[A],
          B: Ordering[B]): Ordering[(A, B)] = …


implicit def contravariantOrdering[A, B]
(implicit C: Converts[A, B],
          O: Ordering[B]): Ordering[A] =
```

Math proofs vs Scala implicit proofs

A | B => C

A => C
+
B => C

```
implicit def ordered[C <% Comparable[C]]: Ordering[C] = …

implicit def comparatorToOrdering[C]
(implicit cmp: Comparator[C]): Ordering[C] = …
```

Avoid ambiguous implicits!

Math proofs vs Scala implicit proofs

!A

!A => B

A => !B

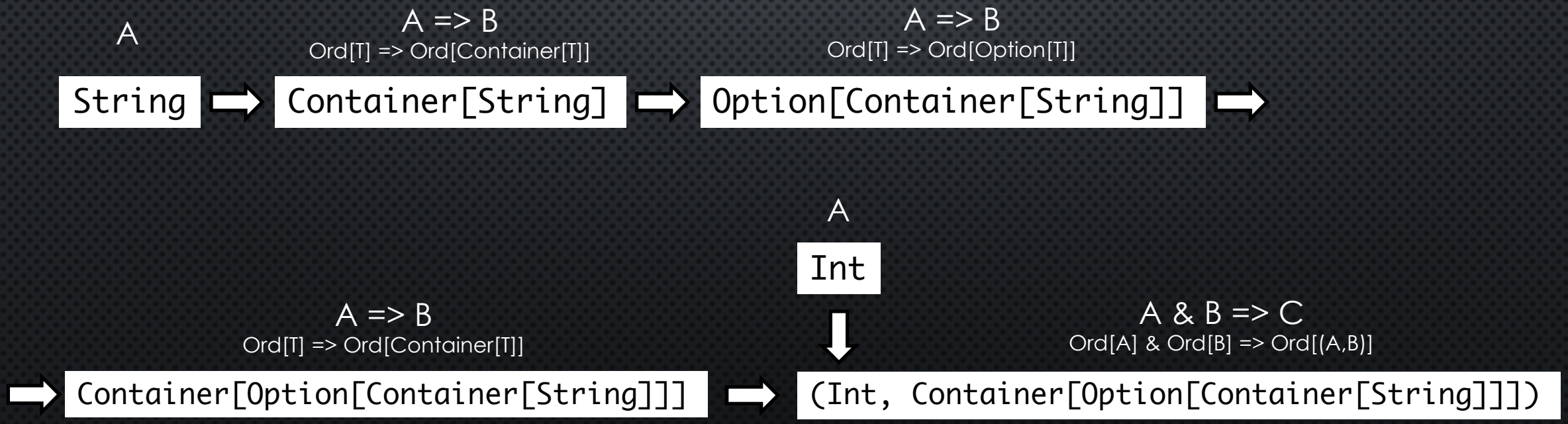
N/A

You can’t ask for an absence of implicit

Back to Ordering example

```
1 -> Container(Option(Container("a")))
```

```
(Int, Container[Option[Container[String]]])
```



HList basics

	Scala List	shapeless.HList
Root type	`List`	`HList`

Cons	`case class ::[A](`
	`  head: A,`
	`  tail: List[A])`
		`case class ::[H, T <: HList](`
		`  head: H,`
		`  tail: T)`

Nil	`Nil` (singleton)	`HNil` (singleton)
Type of `1 :: Nil`	`::[Int]`	`Int :: HNil`
Type of `1 :: "a" :: Nil`	`::[Any]`	`Int :: String :: HNil`
Type of `1 :: "a" :: true :: Nil`	`::[Any]`	`Int :: String :: Boolean :: HNil`

Safe head/tail with precise types

- HList (and HNil) doesn't define head/tail – you just can't call.
- `::` has them both, with precise types:
  - `Int :: String :: HNil` has:
    - `head: Int`
    - `tail: String :: HNil`

Safe indexing

```
final class HListOps[L <: HList](l : L) {

  def at[N <: Nat](implicit at: At[L, N]): at.Out = at(l)
}
```

- Uses type-level natural numbers to build indexing proofs.
- `At[L, N]` is one of that proofs we were talking before.
- If implicit At is found – index is within list bounds.
- If not – index out of range (implicit not found == can't prove).
- Result type is precise (not true for List).

2nd most important property (along with keeping precise types):

Isomorphic to product types

```
case class Person(name: String, age:Int) ⇔ String :: Int :: HNil

(Int, Boolean) ⇔ Int :: Boolean :: HNil
```

- shapeless.Generic[T] typeclass provides conversions between product types' values and HLists
- With the help of implicit macro, Generic instance is available for all tuples and case classes.

HList basics

(Demo)

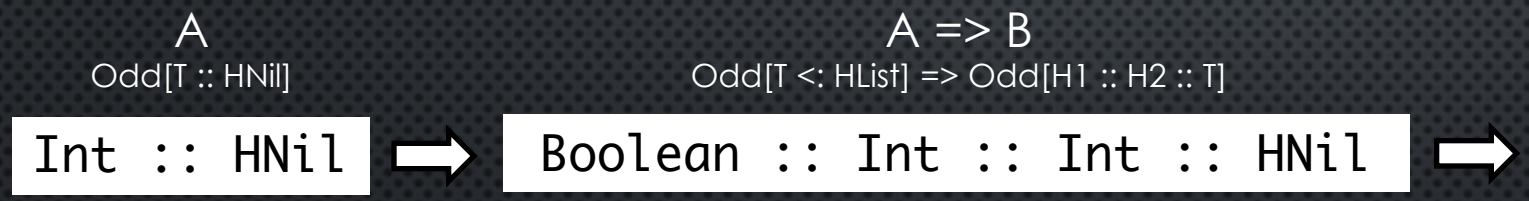
“Pure” HList proof: OddProduct

(Demo)

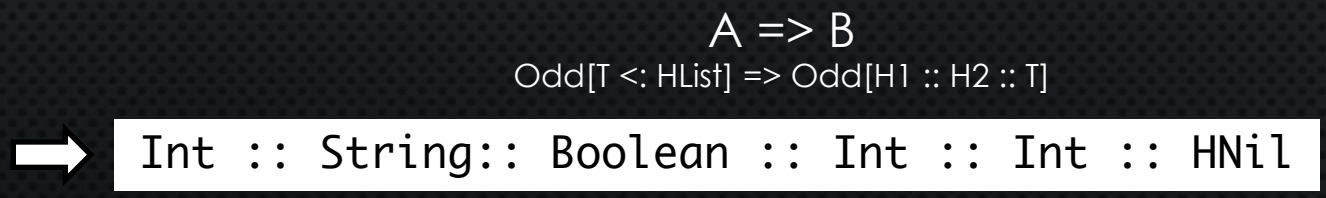
Most of the time HList proofs are inspired by Mathematical Induction

“Pure” HList proof: OddProduct

```
OddProduct[Int :: String :: Boolean :: Int :: Int :: HNil]
```



Recursive step – resolving to the same implicit def



Show typeclass and it's generic proof

(Demo)

Expanding `HList` proof to all product-like types with `shapeless.Generic`

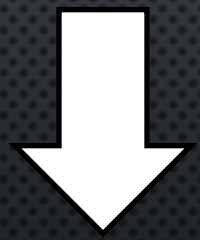
Say we have typeclass Property[A].

If for some type T we have:

`Generic.Aux[T, Repr]` – Isomorphism, that allows us to get `Repr` from `T` and vice versa.
`Repr` – some generic representation of `T`

&

`Property[Repr]` – our `Property` is proved for type `Repr`.



We can (almost always) derive the `Property[T]` by converting to/from `Repr` and using `Property[Repr]`

Hint:
Repr here is going to be an HList

Looks similar to A & B => C.

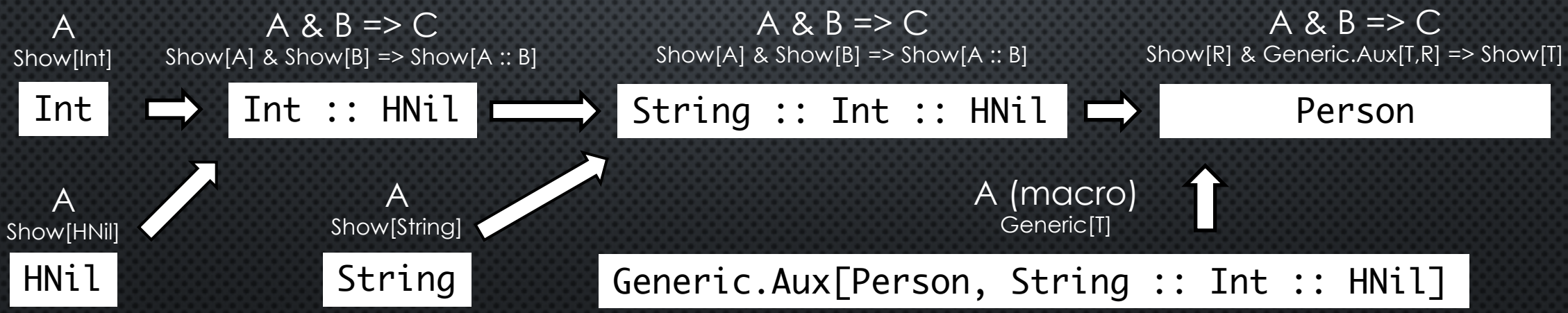
Let's encode the proof!

Expanding `HList` proof to all product-like types with `shapeless.Generic`

(Demo)

Expanding `HList` proof to all product-like types

```
Person(name: String, age: Int)
```



You can automatically derive typeclass instances for any case class with zero boilerplate!

(given the property can be derived from the "shape" of the case class)

Usage examples:

1. Typesafe serialization (e.g. circe JSON)
2. Typesafe conversions between types
3. Generic data diffs (e.g. for API testing)
4. Tuple23, Tuple24, ...
5. ...

Aux Pattern

(Demo)

Generic Product Diffs

(Demo)

What's next?

1. Type-level computations along the proof chain
2. Implicit prioritisation – control potentially ambiguous implicits
3. `shapeless.Coproduct` as a generic representation of sum types (sealed traits)
4. `LabelledGeneric[A]`: Generic, that knows about names of fields and types.
5. Full-blown JSON (de)serialization proof for arbitrary ADT hierarchies
6. Generic transformations with `Poly`

Thank you!