Typeclass derivation with SchemaZ: EZ Katka!

@ValentinKasas (47 Degrees)

ScalaConf 2019

Профессиональная конференция для Scala-разработчиков

Agenda

- Typeclasses
- Derivation
- With SchemaZ

➡ **EZ Katka!**

ScalaConf 2019

Typeclasses: a primer

- Generic interface + Laws
- Implemented by providing a concrete value

Typeclass example

```scala
// typeclass definition
trait PrettyPrint[T] {
  def pprint(t: T): String
}
// typeclass instance for String
implicit object StringPPrint extends PrettyPrint[String] {
  def pprint(str: String): String = "\"" + str + "\""
}
// using the typeclass
class Logger {
  def info[T: PrettyPrint](t: T): Unit = println("INFO " + implicitly[PrettyPrint[T]].pprint(t))
}
```

Aside: different kinds of types

- `*` : the kind of value types (`Int`, `Option[String]`, etc.)
- `* -> *` : type constructors (`List`, `Option`, `Future`, but not `List[_]`)
- A typeclass is always something of kind `k -> *`
    - `trait Monoid[T]` is of kind `* -> *`
    - `trait Functor[F[_]]` is of kind `(* -> *) -> *`

Advantages of typeclasses

- Implementation is independent of type's definition
- Implementations compose
- Laws

Decoupling

- I can implement `TC[T]` independently in a separate file/module/project
- I don't have to own `TC` nor `T`

Composition

- I can build instances using « smaller » ones

```scala
object PPrintInstances {
  implicit def listPPrint[T: PrettyPrint]: PrettyPrint[List[T]] =
    (ts: List[T]) =>
      ts
        .map(implicitly[PrettyPrint[T]].pprint)
        .mkString(", ")
}
```

Laws

- I can reason about the properties of a TC in an algebraic way
  - ➡ I can build proofs and/or PBT
- Therefore I can safely forget about the implementation details

Building apps with typeclasses

- Clean and lean domain model

```scala
sealed trait User
case class Employee(name: String, department: String) extends User
case class Customer(email: String, planId: Long)       extends User
```

- Behaviour added using typeclasses

But dude, that’s still a lot of code to write…

Typeclass derivation

- A way to provide instances automatically
- We'll restrict ourselves to kind $* \to *$ typeclasses

ScalaConf 2019

Derivation libraries

- scalaz-deriving
- shapeless
- magnolia

scalaz-deriving

- Builds upon the algebraic properties of the typeclass (ie. laws)
- Uses TCs like `Applicative`, `Alt`, `Decidable`, `Divisible` (which are of kind `(* -> *) -> *`)
- No access to field names, cannot derive lawless TCs

scalaz-deriving typeclasses

- If you have an `F[A]` and a `F[B]`:
  - `Applicative` gives you an `F[(A, B)]` if `F` is covariant
  - `Alt` gives you an `F[Either[A, B]]` if `F` is covariant
  - `Divisible` gives you an `F[(A, B)]` if `F` is contravariant
  - `Decidable` gives you an `F[Either[A, B]]` if `F` is contravariant

Shapeless

- Based on the algebraic structure of ADTs (sums of products)
- Operations happen at the type level (but there's a value-level API)
- Not easy to test, can blow up compilation times

Magnolia

- Based on structures ADT
- Simple interface in terms of CaseClass and SealedTrait (but still not that intuitive)
- Better compile-time performance
- Not easily tested either

Enter SchemaZ

Meet SchemaZ

- Leans toward the ADT-first approach
- But somehow unifies both approaches
- Provides (arguably much) more than TC derivation
- https://github.com/spartanz/schemaz

SchemaZ overview

- A schema abstraction
- A derivation mechanism (based on recursion schemes)
- A mechanism to express schema evolution (out of our current scope)

SchemaZ architecture

- A zero-dependency core module w/ schema definition and derivation mechanism
- Compat modules for popular typeclasses
- A « generic » module for scrapping even more boilerplate

SchemaZ: a dead-simple algebra

- Every possible (non-recursive) ADT can be represented using just:
  - `Unit`
  - `Either` (binary sum)
  - `Tuple2` (binary product)

Simple algebra

```scala
type MyBit = Either[Unit, Unit]

type MyByte = (Bit, (Bit, (Bit, (Bit, (Bit, (Bit, (Bit, Bit)))))))

type MyOption[A] = Either[Unit, A]
```

Isomorphisms: bridging the gap

- Isomorphisms allow us to tie our simple representation to concrete types

```
val bit2Boolean = Iso[Either[Unit, Unit], Boolean]
  { bit => bit.fold(_ => true, _ => false)}
  { bool => if(bool) Left(()) else Right(())}
```

Creating a Schema

- Let's imagine a business ADT

```scala
sealed trait Role
case class User(active: Boolean)        extends Role
case class Admin(rights: List[String]) extends Role

case class Person(name: String, role: Option[Role])
```

Creating a schema for a case class

```scala
case class Person(name: String, role: Option[Role])

def personSchema(r: SchemaZ[Role]) = caseClass(
  "name" -*>: prim(StringSchema) :*: "role" -*>: optional(r),
  Iso[(String, Option[Role]), Person]
    { (n, r) => Person(n, r)}
    { p => (p.name, p.role)}
)
```

Creating a schema for a sealed trait

```scala
sealed trait Role
final case class User(active: Boolean)       extends Role
final case class Admin(rights: List[String]) extends Role

def roleSchema(u: SchemaZ[User], a: SchemaZ[Admin]) = sealedTrait(
  "user" -+>: u :+: "admin" -+>: a,
  Iso[Either[User, Admin], Role]
    {
      case Left(u)  => u
      case Right(a) => a
    }
    {
      case u @ User(_)  => Left(u)
      case a @ Admin(_) => Right(a)
    }
)
```

Algebra encoding (simplified to fit on a slide)

```scala
sealed trait SchemaF[F[_], A]

case class One[F[_]]()                                     extends SchemaF[F, Unit]
case class Primitive[F[_], A](prim: Prim[A])               extends SchemaF[F, A]
case class Sum[F[_], A, B](left: F[A], right: F[B])        extends SchemaF[F, Either[A, B]]
case class Prod[F[_], A, B](left: F[A], right: F[B])       extends SchemaF[F, (A, B)]
case class Field[F[_], A](name: String, field: F[A])       extends SchemaF[F, A]
case class Record[F[_], A](fields: F[A])                   extends SchemaF[F, A]
case class Union[F[_], A](fields: F[A])                    extends SchemaF[F, A]
case class Sequence[F[_], A](items: F[A])                  extends SchemaF[F, A]
```

- It's just a type of kind (* -> *) -> * -> *, what's the problem?

ScalaConf 2019

SchemaZ definition (again, simplified)

```scala
// a schema representing the type A
trait SchemaZ[A]{
  type R                                // the algebraic representation

  def structure: Fix[SchemaF, R]  // reification of the repr.
  def iso: Iso[R, A]                    // the "glue"
}
```

Err… what is that Fix thing?

```
case class Fix[F[_[_], _], A](unFix: F[Fix[F, ?], A])
```

- Looks a bit scary doesn't it? ((* -> *) -> * -> *) -> * -> *
- Just something we need to use recursion schemes

Fixed-point types

```scala
type F[A] = ???

val unfixed: SchemaF[SchemaF[F, ?], Either[Unit, Unit]] =
  Sum[F, Either[Unit, Unit]](One[F](), One[F]())

val fixed: Fix[SchemaF[F, ?], Either[Unit, Unit]]] =
  Fix(Sum(Fix(One[F]()), Fix(One[F]())))
```

Recursion schemes 101

- A technique to abstract over recursion
- Uses three « ingredients » :
  - a pattern functor (here `SchemaF`)
  - a fixed-point type (over that pattern functor)
  - an Algebra and/or a Coalgebra

Higher order recursion shemes

- Our pattern functor is in fact of an higher kind

```scala
trait HFunctor[H[_[_], _]] {
  def hmap[F[_], G[_]](nt: F ~> G): H[F, ?] ~> H[G, ?]
}
```

- Everything else is as usual, but with natural transformations instead of mere functions

Deriving instances

- Picking a scheme (in 90% of the cases, simply `cata`)
- Writing the required (co)algebra
- Wrapping that up in an implicit `Interpreter`

Example 1: org.scalacheck.Gen

```scala
implicit def genInterpreter(implicit prim2Gen: Prim ~> Gen): Interpreter[Gen] =
  Interpreter.cata(new (SchemaF[Gen, ?] ~> Gen) {
    def apply[A](schema: SchemaF[Gen, A]): Gen[A] = schema match {
      case Primitive(prim)    => prim2Gen(prim)
      case Prod(left, right)  => for (l <- left; r <- right) yield (l, r)
      case Sum(left, right)   => Gen.oneOf(left.map(Left(_)), right.map(Right(_)))
      case Record(fields)     => fields
      case Sequence(element)  => Gen.listOf(element)
      case Field(_, base)     => base
      case Union(choices)     => choices
      case Branch(_, base)    => base
      case One()              => Gen.const(())
    }
  })
```

Usage

- Once we have an implicit interpreter available, it is as simple as:

```
val personGen: Gen[Person] = personSchema.to[Gen]
```

ScalaConf 2019

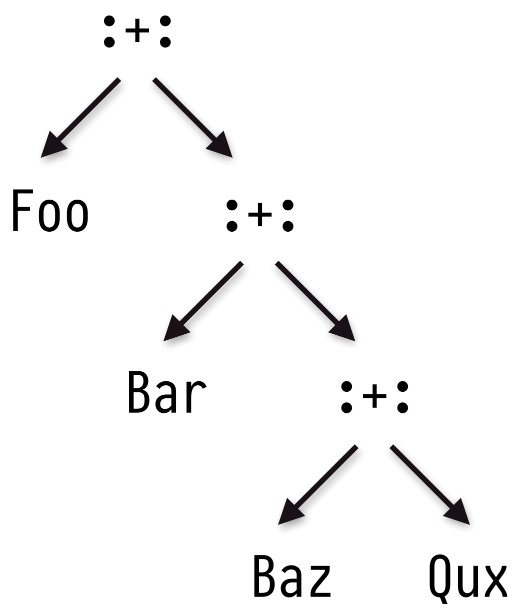
EZ Katka!

Hold-on, it's not always that simple

- Can you spot a bug in our derived `Gen`?
- Think hard, the devil is in the details 😈

The downside of binary sum

- Our sums are nested
- If each branch as the same probability, we'll end up with:
  - p(Foo): 0.5
  - p(Bar): 0.25
  - p(Baz): 0.125
  - p(Qux): 0.125



Thinking outside the box

- Gen has a frequency method

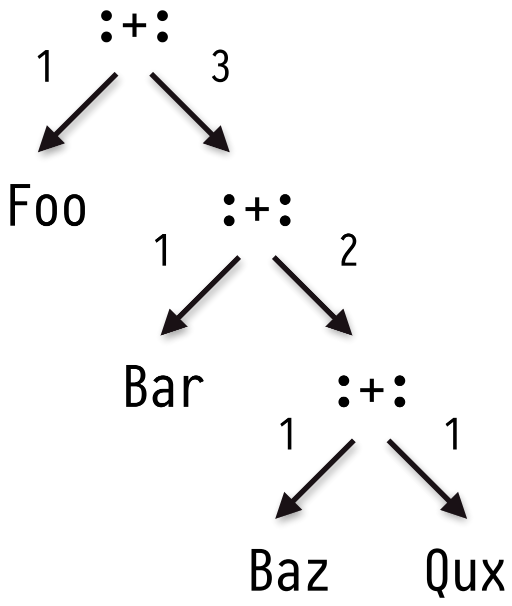
```scala
val oneBoomInThree = Gen.frequency(
  2 -> "you're safe",
  1 -> "boom"
)
```

- So why not use that to our advantage

Fixing our derivation

```scala
// a "weighted Gen"
type WGen[A] = (Int, Gen[A])

// draft of your rewritten algebra
def apply[A](schema: RSchema[WGen, A]): WGen[A] =
  schema match {
    case Primitive(prim) => (1, prim2Gen(prim))
    // Same with all other cases: we wrap the result Gen in a pair
    // The only case that changes is the sum
    case Sum((wl, left), (wr, right)) =>
      (wl + wr, Gen.frequency(
        wl -> left.map(Left(_)),
        wr -> right.map(Right(_))
      ))
  }
```



Unifying with the TC-first approach

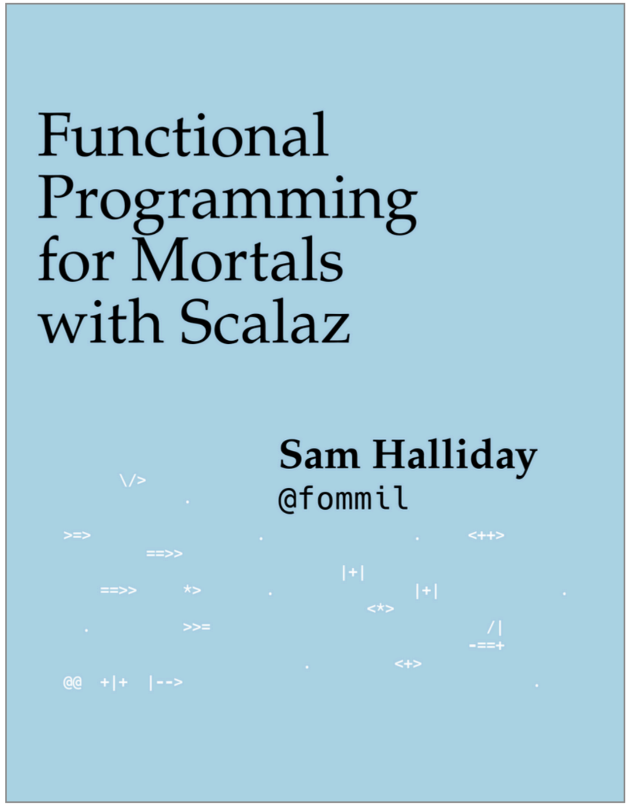
```
def covariantTargetFunctor[H[_]](
    primNT: Prim ~> H,
    seqNT: H ~> λ[X => H[List[X]]]
  )(implicit H: Alt[H]): HAlgebra[SchemaF, H] =
    new (SchemaF[H, ?] ~> H) {

      def apply[A](schema: SchemaF[H, A]): H[A] =
        schema match {
          case Primitive(prim)    => primNT(prim)
          case x: Sum[H, a, b]    => H.either2(x.left, x.right)
          case x: Prod[H, a, b]   => H.tuple2(x.left, x.right)
          case x: Record[H, a]    => x.fields
          case x: Sequence[H, a]  => seqNT(x.element)
          case pt: Field[H, a]    => pt.field
          case x: Union[H, a]     => x.choices
          case st: Branch[H, a]   => st.branch
          case _: One[H]          => H.pure(())
        }
    }
```

- We can define completely TC-generic derivations in terms of `Alt`/`Decidable`

Digging deeper

- Read chapter 8 of « **Functional Programming for Mortals** » (actually read the whole thing, it's great!)
- Look at the actual SchemaZ code
- Ask me anything (`@ValentinKasas`, my DM are open)



ScalaConf 2019

Thank you!

Speaking of which...

- Take a few seconds to participate to the #ScalaThankYou campaign!

➡ https://danielasfregola.com/2019/11/25/introducing-scalathankyou-be-a-part-of-it/

I'm @ValentinKasas

Solution architect
@ 47 Degrees

47

We're Hiring!

https://47deg.com

Спасибо!