
Type
Classes
In Scala


Type
Classes
In ~~Scala~~


Type
Classes
In **Scato**

Scato: An exploration of purely functional library design in Scala

- Experiments for a future Scalaz 8
- Aim at improving encoding of popular FP concepts
- Most notably: Type classes!
- **github.com/aloiscochard/scato**

Type Classes are Great!

- Ad’hoc polymorphism
- for abstractions
- and code reuse
- Come with **laws**
- but not native in Scala.

Type Classes are implicitely dispatched by Type

```
sort :: (Ord a) => [a] -> [a]
```

```
def sort[A: Ord]: List[A] => List[A]
```

```
def sort[A: Ord](l: List[A]): List[A]

def sort[A](l: List[A])(implicit O: Ord[A]): List[A]
```

Resolution of implicit in scala obey complex rules...

Coherent resolution

of implicit type class instance is paramount to preserve equational reasonning.

For a any type “A”, an implicit “**Ord[A]**“ must always resolve to a globally **unique** type class instance, **wherever we are**!

with identity refering to **dynamic semantic**

Enforcing Coherant Resolution

TWO principles:

#1 Any given type class instance “Foo[Bar]” must be able to be resolved implicitly by only importing “Foo” and “Bar”.

So, in practice, **we must define “Foo[Bar]” instances either in the companion object of “Foo” or in the one of “Bar”**.

Enforcing Coherant Resolution

And because type classes come in hierarchy:

#2 Implicit resolution of instance should never result in ambiguities (compile error):

For all type “A” and type class hierarchy “Foo”, there must be an unambiguous way to implicitly get an “Foo[A]” from any instance in scope that **extends** Foo[A].

So, what is the problem?

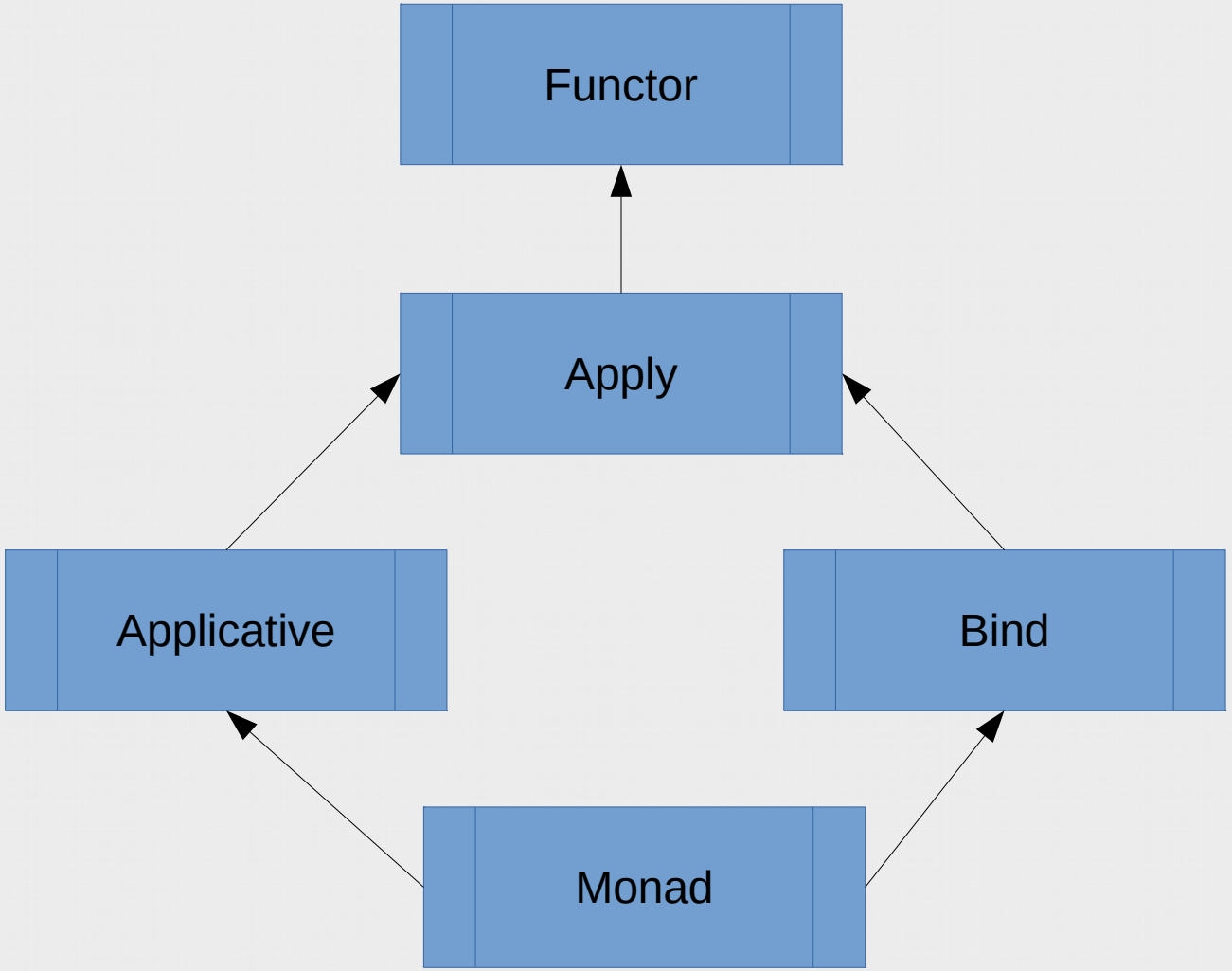
Since scala does not know the concept of type class (and the implied coherence of type class hierarchy), implicit resolution can lead to unresolved ambiguities (compile error).

Why ? Because type class hierarchy is traditionally encoded via

Subtyping

Always there to ruin inference!

Case study: a Functor hierarchy



Hierarchy encoded via Subtyping (1/3)

```scala
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

trait Apply[F[_]] extends Functor[F] {
  def ap[A, B](fa: F[A])(f: F[A => B]): F[B]
}

trait Applicative[F[_]] extends Apply[F] {
  def pure[A](a: A): F[A]
  override def map[A, B](fa: F[A])(f: (A) => B): F[B] = ap(fa)(pure(f))
}

trait Bind[F[_]] extends Apply[F] {
  def flatMap[A, B](ma: F[A])(f: A => F[B]): F[B]
  override def ap[A, B](fa: F[A])(f: F[A => B]): F[B] = flatMap(f)(map(fa))
}

trait Monad[F[_]] extends Applicative[F] with Bind[F] {
  override def map[A, B](fa: F[A])(f: (A) => B): F[B] = flatMap(fa)(a=>pure(f(a)))
}
```

Hierarchy encoded via Subtyping (2/3)

```scala
// Instances
trait MonadInstancesStd {
  implicit val list: Monad[List] = new Monad[List] {
    override def flatMap[A, B](xs: List[A])(f: A => List[B]) = xs.flatMap(f)
    override def pure[A](a: A): List[A] = List(a)
  }

}
// Companion objects
object Functor extends MonadInstancesStd {
  def apply[F[_]](implicit F: Functor[F]): Functor[F] = F
}
object Apply extends MonadInstancesStd{
  def apply[F[_]](implicit F: Apply[F]): Apply[F] = F
}
object Applicative extends MonadInstancesStd{
  def apply[F[_]](implicit F: Applicative[F]): Applicative[F] = F
}
object Bind extends MonadInstancesStd {
  def apply[F[_]](implicit F: Bind[F]): Bind[F] = F
}
object Monad extends MonadInstancesStd {
  def apply[F[_]](implicit F: Monad[F]): Monad[F] = F
}
```

Hierarchy encoded via Subtyping (3/3)

```
def u[A, F[_] : Functor](fa: F[A]) = Functor[F].map(fa)(_ => ())


def x[A, F[_] : Applicative : Bind](fa: F[A]): F[Unit] = u(fa)

Error: ambiguous implicit values:                                  ^^^
  both value evidence$3 of type Bind[F]
and value evidence$2 of type Applicative[F]
match expected type Functor[F]
```

KaBOOM !! :(

Scala does not know that both Bind[F] and Applicative[F] imply the **same** Functor[F] and want to us to make an explicit choice!

Experiment 1: Invariant wrapper (1/3)

```scala
class TC[T[_], C[_[_]]](val instance: C[T]) extends AnyVal

object TC {
  def apply[T[_], C[_[_]]](i: C[T]): TC[T, C] = new TC(i)
}

trait MonadInstancesStd {

  implicit val list: TC[List, Monad] = TC(new Monad[List] {
    override def flatMap[A, B](xs: List[A])(f: A => List[B]): List[B] = xs.flatMap(f)
    override def pure[A](a: A): List[A] = List(a)
  })
}
```

We wrap our instances in a newtype `TC[T[_], C[_[_]]]` that **hide the subtyping relation** and give use the opportunity to override the default implicits resolution mechanism.

Experiment 1: Invariant wrapper (2/3)

Controlling implicit resolution:

```
trait H0 extends H1 {
  implicit def monadBind[F[_]](implicit F: TC[F, Monad]): TC[F, Bind] = TC(F.instance)
  implicit def monadApplicative[F[_]](implicit F: TC[F, Monad]): TC[F, Applicative] = TC(F.instance)
  implicit def monadApply[F[_]](implicit F: TC[F, Monad]): TC[F, Apply] = TC(F.instance)
  implicit def monadFunctor[F[_]](implicit F: TC[F, Monad]): TC[F, Functor] = TC(F.instance)
}

trait H1 extends H2 {
  implicit def applicativeApply[F[_]](implicit F: TC[F, Applicative]): TC[F, Apply] = TC(F.instance)
  implicit def applicativeFunctor[F[_]](implicit F: TC[F, Applicative]): TC[F, Functor] =
TC(F.instance)
}

trait H2 {
  implicit def applyFunctor[F[_]](implicit F: TC[F, Apply]): TC[F, Functor] = TC(F.instance)
}
```

the H0 > H1 > H2 hierarchy define the implicits priority and affect the resolution: if a match is found in H0, implicit resolution will not look in H1/H2, thus avoiding ambiguities.

Experiment 1: Invariant wrapper (3/3)

```scala
def u[A, F[_]](fa: F[A])(implicit F: TC[F, Functor]) = Functor[F].map(fa)(_ => ())

def x[A, F[_] : TC[?[_], Applicative] : TC[?[_], Bind]](fa: F[A]): F[Unit] = u(fa)
```

YEAH !! It compile!

But a bit verbose/clunky.
Could we do better?

Experiment 2: compositionality is King! (1/3)

```scala
trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
}

trait Apply[F[_]] {
  def functor: Functor[F]
  def ap[A, B](fa: F[A])(f: F[A => B]): F[B]
}

trait Applicative[F[_]] {
  def apply: Apply[F]
  def pure[A](a: A): F[A]
}

trait Bind[F[_]] {
  def apply: Apply[F]
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
}

trait Monad[F[_]] {
  def applicative: Applicative[F]
  def bind: Bind[F]
}

object Monad extends MonadInstancesStd {
  def apply[F[_]](implicit F: Monad[F]): Monad[F] = F
}

trait MonadInstancesStd {
  implicit val listMonad = new Monad[List] {
    override val applicative = new Applicative[List] {
      override val apply: Apply[List] = new Apply[List] {
        override val functor: Functor[List] = new Functor[List] {
          override def map[A, B](xs: List[A])(f: (A) => B) = xs.map(f)
        }
        override def ap[A, B](xs: List[A])(f: List[(A) => B]) = xs.flatMap(a => f.map(_ (a)))
      }
      override def pure[A](a: A): List[A] = List(a)
    }
    override val bind = new Bind[List] {
      override val apply: Apply[List] = applicative.apply
      override def flatMap[A, B](xs: List[A])(f: (A) => List[B]) = xs.flatMap(f)
    }
  }
}
```

Experiment 2: compositionality is King (2/3)

We can also control implicit resolution,

```
trait H0 extends H1 {
  implicit def monadBind[F[_]](implicit F: Monad[F]): Bind[F] = F.bind
  implicit def monadApplicative[F[_]](implicit F: Monad[F]): Applicative[F] = F.applicative
  implicit def monadApply[F[_]](implicit F: Monad[F]): Apply[F] = F.applicative.apply
  implicit def monadFunctor[F[_]](implicit F: Monad[F]): Functor[F] =
F.applicative.apply.functor
}

trait H1 extends H2 {
  implicit def applicativeApply[F[_]](implicit F: Applicative[F]): Apply[F] = F.apply
  implicit def applicativeFunctor[F[_]](implicit F: Applicative[F]): Functor[F] =
F.apply.functor
}

trait H2 {
  implicit def applyFunctor[F[_]](implicit F: Apply[F]): Functor[F] = F.functor
}
```

the same way as in experiment 1.

Experiment 2: compositionality is King (3/3)

```scala
def u[A, F[_] : Functor](fa: F[A]) = Functor[F].map(fa)(_ => ())

def x[A, F[_] : Applicative : Bind](fa: F[A]): F[Unit] = u(fa)
```

YEAH !! It compile!
And we restored the nice syntax of the subtyping encoding.

But the list monad instance definition was kind of painful...
Could we do better?

Experiment 3: Templating (1/2)

```scala
trait FunctorClass[F[_]] extends Functor[F]{
  final def functor: Functor[F] = this
}
trait ApplyClass[F[_]] extends Apply[F] with FunctorClass[F] {
  final def apply: Apply[F] = this
}
trait BindClass[F[_]] extends Bind[F] with ApplyClass[F] {
  final def bind: Bind[F] = this
  // derive ap default implementation
  override def ap[A, B](fa: F[A])(f: F[A => B]): F[B] = flatMap(f)(functor.map(fa))
}
trait ApplicativeClass[F[_]] extends Applicative[F] with ApplyClass[F] {
  final def applicative: Applicative[F] = this
  // derive default map implementation
  override def map[A, B](fa: F[A])(f: (A) => B): F[B] = apply.ap(fa)(pure(f))
}
trait MonadClass[F[_]] extends Monad[F] with BindClass[F] with ApplicativeClass[F] {
  final def monad: Monad[F] = this
  // derive default map implementation
  override def map[A, B](fa: F[A])(f: (A) => B): F[B] =
    bind.flatMap(fa)(a => applicative.pure(f(a)))
}
```

Experiment 3: Templating (2/2)

By using template we restore the simple syntax to create instances (and also better perf):

```
trait MonadInstancesStd {

  // nice minimal definition
   val list: Monad[List] = new MonadClass[List] {
    override def flatMap[A, B](xs: List[A])(f: A => List[B]) = xs.flatMap(f)
    override def pure[A](a: A) = List(a)
  }

  // or full definition
  implicit val listFull: Monad[List] = new MonadClass[List] {
    override def ap[A, B](xs: List[A])(f: List[A => B]) = xs.flatMap(a => f.map(_ (a)))
    override def flatMap[A, B](xs: List[A])(f: A => List[B]) = xs.flatMap(f)
    override def map[A, B](xs: List[A])(f: A => B) = xs.map(f)
    override def pure[A](a: A) = List(a)
  }

}
```

Problems solved!

Open the way to proper MTL-style in Scala and many other uses:
https://github.com/scalaz/scalaz/issues/1110

And some hair will be saved.

But rest assured: Scala can give you plenty of other reasons to pull them out!

So, **thank-you for your attention**!
Code is at
https://github.com/jbgi/scato/tree/playground/playground/src/main/scala/typeclasses

And...

Join the Scalazzi

Help building the next, greatest purely functional library by joining the

Scato Experiment

https://github.com/aloiscochard/scato

preparing the way to **Scalaz8**!

we even have a Code of Conflict!

Peace, Love, and Referential Transparency