Run Wild, Run Free!

A team's journey over Scala's FP emerging patterns

What is this about?

— Code samples illustrating common issues folks run into with Scala.

Compiled since 2013 with about ~10 teams with 5 - 6 devs per team
+ many conf conversations.

What is this about?

A code centric report on:

— What a few modern Scala based code bases may look like in the wild since 2013.

— How some of those codebases may progress toward a more FP style.

— Hurdles & caveats Scala devs & FP newcomers face when attempting the above.

— Emerging FP Patterns.

— An opening discussion of what we, as a community, can do to make newcomers life easier.

DISCLAIMER: What it's NOT about!

— A critique of your server as a function

— A critique to [scala.concurrent.Future] or any other Future like impl.

Your server as a function

```
import rwrf.utils._
import scala.concurrent.Future

type Service[Req, Rep] = Req => Future[Rep]
```

A lot of current architectures are based on variations of this.

Let's build something

```
type UserId = Long
type AddressId = Long

case class User(userId: UserId, addressId: AddressId)
case class Address(addressId: AddressId)
```

Exceptions

Purposely unsealed to resemble most exceptions in the Throwable hierarchy

```
case class NotFound(msg : String) extends RuntimeException(msg)
case class DuplicateFound(msg : String) extends RuntimeException(msg)
case class TimeoutException(msg : String) extends RuntimeException(msg)
case class HostNotFoundException(msg : String) extends RuntimeException(msg)
```

DB backend

```
def fetchRemoteUser(userId: UserId) : User =
  throw NotFound(s"user not found with id : $userId")

def fetchRemoteAddress(addressId: AddressId) : Address =
  throw DuplicateFound(s"address duplicate found")
```

Services

Here is where people start tripping

```
val fetchUser: Service[UserId, User] =
  (userId: UserId) => Future {
    fetchRemoteUser(userId)
  }
```

Services

Let's try again

```
import scala.concurrent.ExecutionContext.Implicits.global

val fetchUser: Service[UserId, User] =
  (userId: UserId) => Future(fetchRemoteUser(userId))

val fetchAddress: Service[AddressId, Address] =
  (addressId: AddressId) => Future(fetchRemoteAddress(addressId))

val fetchUserInfo: Service[UserId, (User, Address)] =
  (userId: UserId) =>
    for {
      user <- fetchUser(userId)
      address <- fetchAddress(user.addressId)
    } yield (user, address)
```

Error handling

What if something goes wrong?

```
val fetchUserInfo: Service[UserId, (User, Address)] =
  (userId: UserId) =>
    for {
      user <- fetchUser(userId) //not found
      address <- fetchAddress(user.addressId) //duplicate found
    } yield (user, address)
```

Error handling

At this point folks branch out and they either

— Attempt to Future#recover for both known and unforeseen exceptions.

— Model known exceptional cases via nested Option, Either, Try...

Error handling

Those who attempt to recover may succeed

```
val result = fetchUserInfo(1L) recover {
  case _ : NotFound => User(1L, 10L)
}
```

Error handling

But if you don't know the impl details and attempt to recover

```
val result = fetchAddress(1L) recover {
  case _ : NotFound => Address(1L)
}
```

Error handling

Recovering becomes a game of guess and match

```
fetchAddress(1L) recover {
  case _ : NotFound => Address(1L)
  case _ : DuplicateFound => Address(1L)
}
```

Error handling

What starts as a trivial piece of code quickly becomes

```
import scala.util.control._

fetchAddress(1L) recover {
  case _ : NotFound => ???
  case _ : DuplicateFound => ???
  case _ : TimeoutException => ???
  case _ : HostNotFoundException => ???
  case NonFatal(e) => ???
}
```

Error handling

With most Future#recover style apps I've seen...

— Http 500 production errors due to uncaught exceptions

— Future#recover is abused for both known/ unforeseen exceptions

— Partial Functions + Unsealed hierarchies = Late night fun!

Error handling

Let's mock the DB again

```
val existingUserId = 1L
val nonExistingUserId = 2L

def fetchRemoteUser(userId: UserId) : User =
  if (userId == existingUserId) User(userId, 10L) else null

def fetchRemoteAddress(addressId: AddressId) : Address =
  Address(addressId)
```

Error handling

Folks realize they need safeguards

```
val fetchUser: Service[UserId, Option[User]] =
  (userId: UserId) => Future(Option(fetchRemoteUser(userId)))

val fetchAddress: Service[AddressId, Option[Address]] =
  (addressId: AddressId) => Future(Option(fetchRemoteAddress(addressId)))
```

Error handling

The issue of having a nested type quickly surfaces

```
val fetchUserInfo: Service[UserId, (User, Address)] =
  (userId: UserId) =>
    for {
      user <- fetchUser(userId)
      address <- fetchAddress(user.addressId)
    } yield (user, address)
```

Error handling

Most folks wrestle with the for comprehension but end up doing:

```
val fetchUserInfo: Service[UserId, Option[(User, Address)]] =
  (userId: UserId) =>
    fetchUser(userId) flatMap {
      case Some(user) => fetchAddress(user.addressId) flatMap {
        case Some(address) => Future.successful(Some((user, address)))
        case None => Future.successful(None)
      }
      case None => Future.successful(None)
    }
```

Error handling

PM hands out new requirements...

We need ALL THE INFO for a User in that endpoint! :O

Error handling

As new requirements are added

```
val fetchUserInfo: Service[UserId, Option[(User, Address, PostalCode, Region, Country)]] =
  (userId: UserId) =>
    fetchUser(userId) flatMap {
      case Some(user) => fetchAddress(user.addressId) flatMap {
        case Some(address) => fetchPostalCode(address.postalCodeId) flatMap {
          case Some(postalCode) => fetchRegion(postalCode.regionId) flatMap {
            case Some(region) => fetchCountry(region.countryId) flatMap {
              case Some(country) =>
                Future.successful(Some((user, address, postalCode, region, country)))
              case None => Future.successful(None)
            }
            case None => Future.successful(None)
          }
          case None => Future.successful(None)
        }
        case None => Future.successful(None)
      }
      case None => Future.successful(None)
    }
```

Error handling

Code starts looking like

```
<img src="../../../../../../../../../../../../../../../../../assets/logo.png" />
```

Error handling

Code starts looking like

```
If optCashReportDay.Value = True Then
  DoCashReportDay
Else
  If optCashReportWeek.Value = True Then
    DoCashReportWeek
  Else
    If optCashReportMonth.Value = True Then
      DoCashReportMonth
    Else
      If optCashReportAnnual.Value = True Then
        DoCashReportAnnual
      Else
        If optBondReportDay.Value = True Then
          DoBondReportDay
          // Goes on forever....
```

Error handling

At this point it's a game of choosing your own adventure:

— It compiles and runs, I don't care! (╯°□°)╯ ‾ ┻━━┻

— We should ask for help, things are getting out of control!

— Someones says the word ☠ Monad Transformers ☠

Error handling

OptionT, EitherT, Validated some of the reasons folks get interested in FP in Scala.

Error handling

Those that survive are happy again that their code looks nicer again

```
import cats.data.OptionT
import cats.instances.future._

val fetchUserInfo: Service[UserId, Option[(User, Address)]] =
    (userId: UserId) => {
      val resT = for {
        user <- OptionT(fetchUser(userId))
        address <- OptionT(fetchAddress(user.addressId))
      } yield (user, address)
      resT.value
    }

val existingUser = fetchUserInfo(existingUserId).await

val nonExistingUser = fetchUserInfo(nonExistingUserId).await
```

Error handling

As they dig deeper in FP land they learn about the importance of ADTs and sealed hierarchies!

```
sealed abstract class AppException(msg : String) extends Product with Serializable
final case class NotFound(msg : String) extends AppException(msg)
final case class DuplicateFound(msg : String) extends AppException(msg)
final case class TimeoutException(msg : String) extends AppException(msg)
final case class HostNotFoundException(msg : String) extends AppException(msg)
```

Error handling

Exceptional cases start showing up in return types.

```
import cats.data.Xor
import cats.syntax.xor._

implicit class FutureOptionOps[A](fa : Future[Option[A]]) {
  def toXor[L](e : L): Future[L Xor A] =
    fa map (_.fold(e.left[A])(x => x.right[L]))
}

val fetchUser: Service[UserId, NotFound Xor User] = {
  (userId: UserId) =>
    Future(Option(fetchRemoteUser(userId)))
      .toXor(NotFound(s"User $userId not found"))
}

val fetchAddress: Service[AddressId, NotFound Xor Address] = {
  (addressId: AddressId) =>
    Future(Option(fetchRemoteAddress(addressId)))
      .toXor(NotFound(s"Address $addressId not found"))
}
```

Error handling

They grow beyond OptionT and start using other transformers.

```
import cats.data.XorT

val fetchUserInfo: Service[UserId, NotFound Xor (User, Address)] = {
  (userId: UserId) =>
    val resT = for {
      user <- XorT(fetchUser(userId))
      address <- XorT(fetchAddress(user.addressId))
    } yield (user, address)
    resT.value
}

val existingUser = fetchUserInfo(existingUserId).await
val nonExistingUser = fetchUserInfo(nonExistingUserId).await
```

Non determinism

Common mistakes when refactoring

```
import org.scalacheck._
import org.scalacheck.Prop.{forAll, BooleanOperators}

def sideEffect(latency: Int): Future[Long] =
    Future { Thread.sleep(latency); System.currentTimeMillis }

def latencyGen: Gen[(Int, Int)] = for {
    a <- Gen.choose(10, 100)
    b <- Gen.choose(10, 100)
} yield (a, b)
```

Non determinism

Effects are sequentially performed

```
val test = forAll(latencyGen) { latency =>
    val ops = for {
        a <- sideEffect(latency._1)
        b <- sideEffect(latency._2)
    } yield (a, b)
    val (read, write) = ops.await
    read < write
}
```

Non determinism

Effects may be executed in the wrong order

```
val test = forAll(latencyGen) { latency =>
    val op1 = sideEffect(latency._1)
    val op2 = sideEffect(latency._2)
    val ops = for {
        a <- op1
        b <- op2
    } yield (a, b)
    val (read, write) = ops.await
    read < write
}
```

What others things are folks reporting?

- Wrong order of Effects (When someone recommends moving)
- Random deadlocks (Custom ExecutionContexts)
- General confusion as to why most combinators require an implicit EC
  when one was already provided to Future#apply.
- A lot of reusable code becomes non reusable because it's inside a Future

Code reuse?

Can we make our code work in the context of other types beside Future?

Abstracting over the return type

As it stands our services are coupled to Future.

```
type Service[A, B] = A => Future[B]
```

Abstracting over the return type

But they don't have to.

```
type Service[F[_], A, B] = A => F[B]
```

Abstracting over the return type

We just need a way to lift a thunk: => A to an F[_]

```
import simulacrum.typeclass

@typeclass trait Capture[F[_]] {
  def capture[A](a: => A): F[A]
}
```

Abstracting over the return type

We'll add one instance per type we are wanting to support

```
import scala.concurrent.ExecutionContext

implicit def futureCapture(implicit ec : ExecutionContext) : Capture[Future] =
  new Capture[Future] {
    override def capture[A](a: => A): Future[A] = Future(a)(ec)
  }
```

Abstracting over the return type

```
import monix.eval.Task
import monix.cats._

implicit val taskCapture : Capture[Task] =
  new Capture[Task] {
    override def capture[A](a: => A): Task[A] = Task.evalOnce(a)
  }
```

Abstracting over the return type

```
import scala.util.Try

implicit val tryCapture : Capture[Try] =
  new Capture[Try] {
    override def capture[A](a: => A): Try[A] = Try(a)
  }
```

Abstracting over the return type

Our services now are parametrized to any F[_] for which a Capture
instance is found.

```
import cats.{Functor, Foldable, Monoid}
import cats.implicits._

implicit class FGOps[F[_]: Functor, G[_]: Foldable, A](fa : F[G[A]]) {
  def toXor[L](e : L): F[L Xor A] =
    Functor[F].map(fa) { g =>
      Foldable[G].foldLeft[A, L Xor A](g, e.left[A])((_, b) => b.right[L])
    }
}
```

Abstracting over the return type

Our services now are parametrized to any F[_] for which a Capture instance is found.

```
class Services[F[_] : Capture : Functor] {

  val fetchUser: Service[F, UserId, NotFound Xor User] = {
    (userId: UserId) =>
    Capture[F]
      .capture(Option(fetchRemoteUser(userId)))
      .toXor(NotFound(s"User $userId not found"))
  }

    val fetchAddress: Service[F, AddressId, NotFound Xor Address] = {
      (addressId: AddressId) =>
      Capture[F]
        .capture(Option(fetchRemoteAddress(addressId)))
        .toXor(NotFound(s"Address $addressId not found"))
    }
}

object Services {
    import cats._
    def apply[F[_] : Capture: Monad: RecursiveTailRecM] : Services[F] = new Services[F]
    implicit def instance[F[_]: Capture: Monad: RecursiveTailRecM]: Services[F] = apply[F]
}
```

Abstracting over the return type

Code becomes reusable regardless of the target runtime

```
val futureS = Services[Future].fetchUser(existingUserId)
val taskS = Services[Task].fetchUser(existingUserId)
val tryS = Services[Try].fetchUser(existingUserId)
```

Abstracting over the return type

Other alternatives available (Rapture Modes and Result)

Abstracting over implementations

Now that we can run our code to any F[_] that can capture a lazy computation we may want to abstract over
implementations too.

Abstracting over implementations

Free Monads / Applicatives is what has worked best for us.

— Free of interpretation allowing multiple runtimes.
— Composable via Coproduct.
— Lots of boilerplate that can be improved.
— Supports abstracting over return types.
— Getting momentum with multiple posts and libs supporting the pattern.
(Freek, cats, ...)

Abstracting over implementations

Let's refactor our services to run on Free?

Abstracting over implementations

Define your Algebra

```
sealed abstract class ServiceOp[A] extends Product with Serializable
final case class FetchUser(userId: UserId) extends ServiceOp[NotFound Xor User]
final case class FetchAddress(addressId: AddressId) extends ServiceOp[NotFound Xor Address]
```

Abstracting over implementations

Lift your Algebra to Free

```
import cats.free.Free

type ServiceIO[A] = Free[ServiceOp, A]

object ServiceOps {

    def fetchUser(userId: UserId): ServiceIO[NotFound Xor User] =
      Free.liftF(FetchUser(userId))

    def fetchAddress(addressId: AddressId): ServiceIO[NotFound Xor Address] =
      Free.liftF(FetchAddress(addressId))

}
```

Abstracting over implementations

Write 1 or many interpreters that may be swapped at runtime

```
import cats._
import cats.implicits._
import Services._

def interpreter[M[_] : Capture : Monad : RecursiveTailRecM]
  (implicit impl: Services[M]): ServiceOp ~> M = new (ServiceOp ~> M) {
    override def apply[A](fa: ServiceOp[A]): M[A] = {
      val result = fa match {
        case FetchUser(userId) => impl.fetchUser(userId)
        case FetchAddress(addressId) => impl.fetchAddress(addressId)
      }
      result.asInstanceOf[M[A]]
    }
  }
```

Abstracting over implementations

Write programs using the smart constructors and combining them at will

```
def fetchUserInfo(userId: UserId): ServiceIO[NotFound Xor (User, Address)] = {
    val resT = for {
        user <- XorT(ServiceOps.fetchUser(userId))
        address <- XorT(ServiceOps.fetchAddress(user.addressId))
    } yield (user, address)
    resT.value
}
```

Abstracting over implementations

Run your programs to any target implementation and runtime

```
val tryResult = fetchUserInfo(existingUserId).foldMap(interpreter[Try])

val taskResult = fetchUserInfo(existingUserId).foldMap(interpreter[Task])

val futureResult = fetchUserInfo(existingUserId).foldMap(interpreter[Future])
```

Patterns

Recommendations for others that have worked for us:

— Algebraic design and sealed hierarchies for safer exceptions control.
(Don't match and Guess).

— Abstract over return types for code reuse.

— Abstract over implementations to increase flexibility and composition.

Conclusion

- Most Scala newcomers are NOT exposed to Typed FP when they start.
- There are repeating patterns of failure and frustration among newcommers.
- Scala is not a Functional Programming Language but we can make it.
- Expose newcomers to Scala to typed FP for a brither Future

Wishes for the Future

— Make Scala more FP friendly.

@raulraja @47deg
https://speakerdeck.com/raulraja/run-wild-run-free

Acknowledgments

— 47 Degrees

— Background photo

— Algebraic Data Types

— Cats

— Rapture

— scala.concurrent.Future

— Your server as a function