Flatten your code

@eamelink / Amsterdam.scala / 5-6-2014

Outline

- Flattening code with for-comprehensions
- Flattening containers with transformers
- Type classes, creating transformers
- Applied to Play

Flattening code

The problem

```
def getUserName(data: Map[String, String]): Option[String]
def getUser(name: String): Option[User]
def getEmail(user: User): String
def validateEmail(email: String): Option[String]
def sendEmail(email: String): Option[Boolean]

val data = Map[String, String]()

// Not great: nested maps and flatMaps!
getUserName(data).map { username =>
  getUser(username).map { user =>
    val email = getEmail(user)
    validateEmail(email).map { validatedEmail =>
      sendEmail(validatedEmail)
    }
  }
}
```

The solution

```
def getUserName(data: Map[String, String]): Option[String]
def getUser(name: String): Option[User]
def getEmail(user: User): String
def validateEmail(email: String): Option[String]
def sendEmail(email: String): Option[Boolean]

val data = Map[String, String]()

// Much better: for comprehensions!
for {
  username <- getUserName(data)
  user <- getUser(username)
  email = getEmail(user)
  validatedEmail <- validateEmail(email)
  success <- sendEmail(email)
} yield success
```

for-comprehensions

are really cool!

Syntactic Sugar

- For comprehension syntactic sugar for map, flatMap, (and foreach, withFilter)
- Works for anything that has these methods
- These methods can even be pimped on!

Let’s upgrade Option to Either

- Option is quite limited, ‘failure’ case is empty, no indication of what went wrong
- Scala has an alternative type, `Either`
  - Right side is ‘right’
  - Left side is ‘failure’
- Doesn’t have `map` or `flatMap` though…
- But you can create a `right projection`, that does!

Using Either instead of Option

```
// Instead of `Option`, we use `Either`
def getUserName(data: Map[String, String]): Either[String,
String]
def getUser(name: String): Either[String, User]
def getEmail(user: User): String
def validateEmail(email: String): Either[String, String]
def sendEmail(email: String): Either[String, Boolean]

val data = Map[String, String]()
```

Using Either instead of Option

```
// But while you can do some things...
for {
  username <- getUserName(data).right
  user <- getUser(username).right
} yield user

// You can't do some other things
for {
  username <- getUserName(data).right
  user <- getUser(username).right
  email = getEmail(user)
  validatedEmail <- validateEmail(email).right
  success <- sendEmail(email).right
} yield success

// Desugars to either.right.map{…}.map{…} and breaks!
```

Oh noes, can't use Either…

- Scala's Either is unbiased
- We want a biased container, one that favors the right side

Scalaz \/

- Scalaz has an either, class \/ and named ‘disjunction’ or ‘either’.
- Instead of Left, it has -\/
- Instead of Right, it has \/-
- You can use infix notation. Instead of Either[Foo, Bar], you can use Foo \/ Bar

Using \/ instead of Option

```
// Instead of `Option`, we use `\/`
def getUserName(data: Map[String, String]): String \/ String
def getUser(name: String): String \/ User
def getEmail(user: User): String
def validateEmail(email: String): String \/ String
def sendEmail(email: String): String \/ Boolean

val data = Map[String, String]()
```

Using \/ instead of Option

```
for {
  username <- getUserName(data)
  user <- getUser(username)
  email = getEmail(user)
  validatedEmail <- validateEmail(email)
  success <- sendEmail(email)
} yield success
```

Multiple container types: problem

```
// This doesn't work
def fa: Option[Int] = ???
def fb: String \/ Int = ???
```

```
for {
  a <- fa
  b <- fb
} yield a + b
```

But we can ‘upgrade’ Option to \/

```
// This does work
def fa: Option[Int] = ???
def fb: String \/ Int = ???

for {
  a <- fa.toRightDisjunction("fa is empty!")
  b <- fb
} yield a + b
```

Shorthand notation…

```scala
// This does work
def fa: Option[Int] = ???
def fb: String \/ Int = ???

for {
  a <- fa \/> "fa is empty!"
  b <- fb
} yield a + b
```

Other way around also possible

- `\/` has a method `toOption` to go the other way.

Exercises!

- Get eamelink/flatten from GitHub
- Read README.md
- Open flatten-basics SBT project in an IDE
- Read parts 1 to 5, and do the exercises!

Flattening containers

So far so good, but…

- Sometimes we're dealing with nested containers!

```
val fa: Future[Option[Int]] = ???
val fb: Future[Option[Int]] = ???

// Problem, `a` and `b` are Option[Int], and not Int!
for {
  a <- fa
  b <- fb
} yield a - b
```

Nested containers

- The for comprehension desugars to `map` and `flatMap` on the Future, but doesn’t get the value out of the Option inside the Future!
- A for comprehension only unwraps a single container
- What to do?!?
- Flatten containers!
- We need a thing that has `map` and `flatMap` methods and that can work with a value inside an Option in a Future.

A custom container for Future[Option[X]]

Meet FutureOption!

```
case class FutureOption[A](contents: Future[Option[A]]) {
  def flatMap[B](fn: A => FutureOption[B]) = FutureOption {
    contents.flatMap {
      case Some(value) => fn(value).contents
      case None => Future.successful(None)
    }
  }

  def map[B](fn: A => B) = FutureOption {
    contents.map { option =>
      option.map(fn)
    }
  }
}
```

What did we do?

- We created a container with `map` and `flatMap` methods, that work on a value inside an Option in a Future.

Exercises!

- Read parts 6 and 7, and do the exercises!

Towards Monad Transformers

Subtype polymorphism

```
// Subtype polymorphism: all types that must be serialized
extend a common trait.


trait Serializable {
  def bytes: Array[Byte]
}

def toBytes(value: Serializable) = value.bytes


// Often impractical, because all classes must extend
Serializable. What if we want to serialize `String` or
`Int`???
```

Ad hoc polymorphism

```
// Ad-hoc polymorphism is also known as function
overloading:

def toBytes(value: String): Array[Byte] = value.getBytes
def toBytes(value: Int): Array[Byte] =
value.toString.getBytes

// Also impractical, because now our serialization library
must know about all possible types we want to serialize.
What about the custom types in our app?
```

```
// Solution: glue objects: an object that knows how to
serialize a single type.
// We can create these for all types we want to serialize,
without needing to change those types.

trait Serializable[A] {
  def serialize[A](value: A): Array[Byte]
}

def toBytes[A](value: A, serializer: Serializable[A]):
  Array[Byte] = serializer.serialize(value)
```

```
val StringSerializable = new Serializable[String] {
  override def serialize(value: String) =
    value.getBytes
}

val IntSerializable = new Serializable[Int] {
  override def serialize(value: Int) =
    value.toString.getBytes
}
```

```scala
// In scala, this can be made nicer by making the glue object implicit:
trait Serializable[A] {
  def serialize[A](value: A): Array[Byte]
}

def toBytes[A](value: A)(implicit serializer: Serializable[A]) =
  serializer.serialize(value)

// Or using a `Context Bound`, which is syntactic sugar for the one above
def toBytes2[A : Serializable](value: A) =
  implicitly[Serializable[A]].serialize(value)

implicit val StringSerializable = new Serializable[String] {
  override def serialize(value: String) = value.getBytes
}

implicit val IntSerializable = new Serializable[Int] {
  override def serialize(value: Int) = value.toString.getBytes
}
```

A type class is an interface, that’s implemented outside the type

- Of course we knew all this!
  - Standard library: Numeric, Ordering
  - Play: Format, Reads, Writes type classes when dealing with JSON

Back to our FutureOption

```
case class FutureOption[A](contents: Future[Option[A]]) {
  def flatMap[B](fn: A => FutureOption[B]) = FutureOption {
    contents.flatMap {
      case Some(value) => fn(value).contents
      case None => Future.successful(None)
    }
  }

  def map[B](fn: A => B) = FutureOption {
    contents.map { option =>
      option.map(fn)
    }
  }
}
```

From `Future`, we only use:

- flatMap
- map
- Creating a new one: Future.successful

Monad

- Monad is an typeclass, with methods:
  - map
  - flatMap
  - create

Back to our FutureOption

```
case class FutureOption[A](contents: Future[Option[A]]) {
  def flatMap[B](fn: A => FutureOption[B]) = FutureOption {
    contents.flatMap {
      case Some(value) => fn(value).contents
      case None => Future.successful(None)
    }
  }

  def map[B](fn: A => B) = FutureOption {
    contents.map { option =>
      option.map(fn)
    }
  }
}
```

So we can generalize FutureOption, to make it work for anything for which we have a Monad type class instance, and not just Futures!

Meet AnyMonadOption

```
case class AnyMonadOption[F[_], A](contents: F[Option[A]])
(implicit monadInstanceForF: Monad[F]) {
  def flatMap[B](fn: A => AnyMonadOption[F, B]) =
AnyMonadOption[F, B] {
    monadInstanceForF.flatMap(contents){
      case Some(value) => fn(value).contents
      case None => monadInstanceForF.create(None)
    }
  }

  def map[B](fn: A => B) = AnyMonadOption[F, B] {
    monadInstanceForF.map(contents){ option =>
      option.map(fn)
    }
  }
}
```

Exercises!

- Read parts 8 to 12, and do the exercises!

But Scalaz did this for us

Scalaz has an `AnyMonadOption`, except:

- It's called Option Transformer
- It's class OptionT

Also, Scalaz Monad looks a bit different:

- map can be implemented with 'flatMap' and 'create', so only 'flatMap' and 'create' are abstract
- flatMap is called 'bind'
- create is called 'point'

Scalaz Monad Transformers

Scalaz OptionT

```
// A small example:
val fa: Future[Option[Int]] = ???
val fb: Future[Option[Int]] = ???

// Here, a and b are Int, extracted from both the Future and
the Option!
val finalOptionT = for {
  a <- OptionT(fa)
  b <- OptionT(fb)
} yield a + b

// And to get back to the normal structure:
val finalFutureOption: Future[Option[Int]] =
  finalOptionT.run
```

Making it look nice

```
// Scalaz has a function application operator, that reverses
function and parameter.
// This:
val y1 = double(5)
// Is equivalent to this:
val y2 = 5 |> double
```

Exercises!

- Read parts 13 and 14, and do the exercises!

Applied to Play

Play Action, without for-comprehension

```
def index = Action.async { request =>
  val data = request.queryString.mapValues(_.head)

  UserService.getUserName(data).map { username =>
    UserService.getUser(username).flatMap {
      case None => Future.successful(NotFound("User not found"))
      case Some(user) => {
        val email = UserService.getEmail(user)
        UserService.validateEmail(email).bimap(
          validatedEmail => {
            UserService.sendEmail(validatedEmail) map {
              case true => Ok("Mail successfully sent!")
              case false => InternalServerError("Failed to send email :(")
            }
          },
          errorMsg =>
Future.successful(InternalServerError(errorMsg))).fold(identity, identity)
      }
    }
  } getOrElse Future.successful(BadRequest("Username missing from data!"))
}
```

Play Action, with for-comprehension

```
def index = Action.async { request =>
  val data = request.queryString.mapValues(_.head)

  val result = for {
    username <- UserService.getUserName(data) \/>
                        BadRequest("Username missing from request") |>
                        Future.successful |> EitherT.apply
    user <- UserService.getUser(username)
                        .map { _ \/> NotFound("User not found") } |>
                        EitherT.apply
    email = UserService.getEmail(user)
    validatedEmail <- UserService.validateEmail(email)
                        .leftMap(InternalServerError(_)) |>
                        Future.successful |> EitherT.apply
    success <- UserService.sendEmail(validatedEmail).map { \/-(_) } |>
                        EitherT.apply
  } yield {
    if(success) Ok("Mail successfully sent!")
    else InternalServerError("Failed to send email :(")
  }

  result.run.map { _.fold(identity, identity) }

}
```

Extracting common code

```
// Type alias for our result type
type HttpResult[A] = EitherT[Future, SimpleResult, A]

// Constructors for our result type
object HttpResult {
  def point[A](a: A): HttpResult[A] = EitherT(Future.successful(\/-(a)))
  def fromFuture[A](fa: Future[A]): HttpResult[A] =
    EitherT(fa.map(\/-(_)))
  def fromEither[A](va: SimpleResult \/ A): HttpResult[A] =
    EitherT(Future.successful(va))
  def fromEither[A, B](failure: B => SimpleResult)(va: B \/ A):
    HttpResult[A] = EitherT(Future.successful(va.leftMap(failure)))
  def fromOption[A](failure: SimpleResult)(oa: Option[A]):
    HttpResult[A] = EitherT(Future.successful(oa \/> failure))
  def fromFOption[A](failure: SimpleResult)(foa: Future[Option[A]]):
    HttpResult[A] = EitherT(foa.map(_ \/> failure))
  def fromFEither[A, B](failure: B => SimpleResult)
    (fva: Future[B \/ A]): HttpResult[A] =
    EitherT(fva.map(_.leftMap(failure)))
}

// Converter from our result type to a Play result
def constructResult(result: HttpResult[SimpleResult]) = result.run
  .map { _.fold(identity, identity) }
```

Play action, common code extracted

```
def index = Action.async { request =>
  val data = request.queryString.mapValues(_.head)

  val serviceResult = for {
    username <- UserService.getUserName(data) |>
                HttpResult.fromOption(
                  BadRequest("Username missing from request"))
    user <- UserService.getUser(username) |>
                HttpResult.fromFOption(NotFound("User not found"))
    email = UserService.getEmail(user)
    validatedEmail <- UserService.validateEmail(email) |>
                HttpResult.fromEither(InternalServerError(_))
    success <- UserService.sendEmail(validatedEmail) |>
                HttpResult.fromFuture
  } yield {
    if(success) Ok("Mail successfully sent!")
    else InternalServerError("Failed to send email :(")
  }

  constructResult(serviceResult)
}
```

No More Exercises!

- But you can see this in action in parts 15 to 17. These are in the `flatten-play` directory, controllers package.

“A monad is just a monoid in the category of endofunctors, what's the problem?”

–Mac Lane, sort of.