ScalaSyd

Episode 56 (13 July 2017)

Monadic Matching Mishaps

Mark Hopkins

@antiselfdual | mjhopkins

Let's write some code

```
object DB {
  def loadUser(id: Int): Error \/ (Name, Age)
}
```

```scala
import scalaz.\/
import scalaz.syntax.either._

type Age   = Int
type Name  = String
type Error = String
```

```scala
object DB {
  def loadUser(id: Int): Error \/ (Name, Age)
}
```

```scala
object DB {
  def loadUser(id: Int): Error \/ (Name, Age) =
    id match {
      case 3 => ("Alice", 32).right
      case 5 => ("Bob", 22).right
      case _ => "User not found".left
    }
  }
```

Let's use this in another simple function.

```
def printUser(id: Int): Error \/ String =
  for {
    (name, age) <- DB.loadUser(id)
  } yield s"User $name is $age years old"
```

SYNTAX ERROR 

```
[error] could not find implicit value
  for parameter M: scalaz.Monoid[String]
[error]     (name, age) <- DB.loadUser(id)
                                       ^
```

WTF?! 😱

If we break it up into two lines, it works

```
def printUser(id: Int): Error \/ String =
  for {
    tuple        <- DB.loadUser(id)
    (name, age) = tuple
  } yield s"User $name is $age years old"
```

Compiles 😀

If we replace `Error \/` with `Option`, it works

```
def loadUser(id: Int): Option[(Name, Age)] = ...

def printUser(id: Int): Option[String] =
  for {
    (name, age) <- DB.loadUser(id)
  } yield s"User $name is $age years old"
```

Compiles 😀

If we desugar, it works

```
def printUser(id: Int): Error \/ String =
  DB.loadUser(id) map {
    case (name, age) => s"User $name is $age years old"
  }
```

Compiles 😀

How does `for` *really* desugar?

Hint: it's not just maps and flatMaps

We can use Li Haoyi's **Ammonite REPL** to help us find out.

http://ammonite.io

```
desugar {
  for {
    (name, age) <- DB.loadUser(id)
  } yield s"User $name is $age years old"
}
```

gives

```
    DB.loadUser(id)
      .withFilter {
        case (_, _) => true
        case _      => false
      }
      .map {
        case (name, age) =>
          StringContext.apply("User ", " is ", " years old")
           .s(name, age)
      }
```

😯

There *was* also a warning that might have helped us guess this.

```
[warn] `withFilter' method does not yet exist on
scalaz.\/[Error,(String, Int)], using `filter' method instead
[warn]     (name, age) <- DB.loadUser(id)
```

Scala has decided that

```
(name, age) <- DB.loadUser(id)
```

is a **pattern-match**.

This would be

```
DB.loadUser(id) match { case (name, age) => ... }
```

if we weren't in the middle of a for expression.

In order to do *any* pattern match inside a **for** expression, we need to provide either `filter` or `withFilter`.

What *is* the `filter` method on `\/`?

```scala
/** Filter on the right of this disjunction. */
def filter[AA >: A](p: B => Boolean)
  (implicit M: Monoid[AA]): AA \/ B =

  this match {
    case -\/(_) => this
    case \/-(b) => if (p(b)) this else -\/(M.zero)
  }
```

If we *don't* match, then we have to provide something.

The `Monoid` constraint is there to provide that something.

```
trait Monoid[F] {
  def zero: F
  def append(f1: F, f2: => F): F
}
```

In this case we only care about `zero`.

`Monoid` really is overkill.

(Scalaz *used to* have a `Pointed[_]` type class...)

This `filter` / `withFilter` makes more sense in the case when the pattern match isn't guaranteed to succeed (**irrefutable**).

i.e. is an unsafe match that could fail,
as it doesn't handle all possible cases.

```
def sayHi(lists: List[List[String]]) =
  for {
    List(title, name) <- lists
  } yield s"Greetings, $title $name"
```

```scala
def sayHi(lists: List[List[String]]) =
  for {
    List(title, name) <- lists
  } yield s"Greetings, $title $name"
```

```
> sayHi(List(
  List("Lady", "Ada Lovelace"),
  List("Mark"),
  List("President", "Obama"),
  List("Sir", "David Attenborough")
))

res0: List[String] = List(
 "Greetings, Lady Ada Lovelace",
 "Greetings, President Obama",
 "Greetings, Sir David Attenborough"
)
```

The "zero" for lists is empty list, so this makes sense.

The problem is that Scala doesn't properly distinguish between refutable and irrefutable pattern matches.

It **demands** we be able to filter our monad any time a pattern match is present.

Even when the pattern match is guaranteed to succeed.

e.g. matching a tuple `val (x, y) = (3, "hi")`

To make things worse, `filter` doesn't make sense for many monads.

e.g. `A \/ B` in the cases where `A` isn't naturally a monoid.

Are we doomed, or can we fix it?

I'll tell you in a minute!

First, let's take a scenic tour of some other languages that have both

- pattern-matching
- monads

to see how they compare.

idris-lang.org

Idris is a general purpose pure functional programming language with dependent types.

Functions can be either total or partial.

A total function requires that all cases are handled.

```
total
printUser : Int -> Either Error String
printUser id = do
  (name, age) <- DB.loadUser id
  pure $ "User " ++ name ++ " is " ++ show age ++ " years old"
```

Compiles 😀

Idris recognises that matching a tuple can't fail.

```
total
sayHi : List (List String) -> List String
sayHi lists = do
  [title, name] <- lists
  pure $ "Greetings, " ++ title ++ " " ++ name
```

Compilation failure 😢

```
sayHi is possibly not total due to:
    case block in sayHi at <file>.idr:12:20,
    which is not total as there are missing cases
```

But this is true.

So this is actually great:

Idris recognises that this match is not guaranteed to succeed.

To fix

Either

- change `total` to `partial`.
  But then we're saying it's okay to blow up at runtime 💣🔥
- or (better) add the missing cases.
  Idris provides a nice syntax for this.

```
total
sayHi : List (List String) -> List String
sayHi lists = do
  [title, name] <- lists
    | fname :: _ => ["Oy " ++ fname ++ ", bugger off!"]
    | []         => []
  pure $ "Greetings, " ++ title ++ " " ++ name
```

Compiles 😀

and runs:

```
["Greetings, Lady Ada Lovelace",
 "Oy Mark, bugger off!",
 "Greetings, President Obama",
 "Greetings, Sir David Attenborough"] : List String
```

https://www.haskell.org/

Haskell is a general purpose pure functional programming language.

```haskell
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

```haskell
do                  >>>  f >>= (\a ->
  a <- f            >>>    g a >>= (\b ->
  b <- g a          >>>      return (a, b)
  return (a, b)     >>>    )
                    >>>  )
```

Haskell's fine with our tuple-matching case.

```haskell
printUser :: Int -> Either Error String
printUser id = do
  (name, age) <- loadUser id
  pure $ "User " ++ name ++ " is " ++ show age ++ " years old"
```

Compiles 😀

When we give it an unsafe pattern match it compiles...

```
sayHi :: Monad m => m [String] -> m String
sayHi lists = do
  [title, name] <- lists
  return $ "Greetings, " ++ title ++ " " ++ name
```

It seems fine with lists

```
λ> sayHi [["Lady", "Ada Lovelace"], ["Mark"], ["President",
  "Obama"], ["Sir", "David Attennborough"]]
["Greetings, Lady Ada Lovelace","Greetings, President Obama",
"Greetings, Sir David Attennborough"]
```

but an either blows up when the match fails:

```
λ> sayHi (Right ["Lady", "Ada Lovelace"])
Right "Greetings, Lady Ada Lovelace"

λ> sayHi (Right ["Mark"])
*** Exception: Pattern match failure in do expression at ...
```

What's going on?

```
class Monad m where
 ...
 fail :: String -> m a
```

> Fail with a message.
>
> This operation is not part of the mathematical definition of a monad, but is invoked on pattern-match failure in a do expression.

In fact, `<-` always desugars like this:

```
do pat <- computation          >>> let f pat = more
  more                         >>>     f _ = fail "Pattern match
                                         failure in do expression"
                               >>> in  computation >>= f
```

> The problem with this is that fail cannot (!) be sensibly implemented for many monads, for example Either, State, IO, and Reader. In those cases it defaults to `error`.
>
> As a consequence, in current Haskell, you can not use `Monad` polymorphic code safely, because although it claims to work for all `Monad`, it might just crash on you. This kind of implicit non-totality baked into the class is terrible.

Yes, yes it is

Enter the `MonadFail` proposal

A new type class

```
class Monad m => MonadFail m where
    fail :: String -> m a
```

`{-# LANGUAGE -XMonadFailDesugaring #-}` (since GHC 8.0)

Now pattern-matching in monads tries harder.

> A `Monad` without a `MonadFail` instance may only be used in conjunction with pattern that always match, such as newtypes, tuples, data types with only a single data constructor, and irrefutable patterns (~pat).

The safe match still works. 👍

```haskell
sayHi :: Monad m => m [String] -> m String
  sayHi lists = do
    [title, name] <- lists
    return $ "Greetings, " ++ title ++ " " ++ name
```

```
error:
    • Could not deduce (Control.Monad.Fail.MonadFail m)
        arising from a do statement
        with the failable pattern ‘[title, name]’
      from the context: Monad m
        bound by the type signature for:
                   sayHi :: Monad m => m [String] -> m String
        at <interactive>:33:1-42
      Possible fix:
        add (Control.Monad.Fail.MonadFail m) to the context of
          the type signature for:
            sayHi :: Monad m => m [String] -> m String
```

Better 👍

Purescript is a small strongly typed language that compiles to Javascript.

It's okay with the safe pattern match 😀

```
printUser :: Int -> Either Error String
printUser id = do
  Tuple name age <- loadUser id
  pure $ "User " <> name <> " is " <> show age <> " years old"
```

Despite being a functional programming language Purescript doesn't have tuples 😱

(there is a record *called* `Tuple` in `purescript-tuples`)

Let's see how it handles the unsafe match.

```
sayHi :: Array (Array String) -> Array String
sayHi lists = do
  [title, name] <- lists
  pure $ "Greetings, " <> title <> " " <> name
```

Compilation failure

```
Error found:

A case expression could not be determined to cover
all inputs.

  The following additional cases are required
  to cover all inputs:

    _

  Alternatively, add a Partial constraint to the
  type of the enclosing value.
```

```
sayHi :: Partial => Array (Array String) -> Array String
sayHi lists = do
  [title, name] <- lists
  pure $ "Greetings, " <> title <> " " <> name
```

```haskell
sayHi :: Array (Array String) -> Array String
sayHi lists = do
  list <- lists
  case list of
    [title, name] -> pure $ "Greetings, " <>
                       title <> " " <> name
    [name]        -> pure $ "Oy " <> name <> " bugger off!"
    _             -> []
```

Back to

The problem is that Scala doesn't properly distinguish between refutable and irrefutable pattern matches.

It **demands** we be able to filter our monad any time a pattern match is present.

Even when the pattern match is guaranteed to succeed.

e.g. matching a tuple

Are we doomed or can we fix it?

Pimps* to the rescue

* implicit enhancements


PIMPS* TO
THE RESCUE
IMPLICIT
ENHANCEMENTS
imgflip.com

```
object DisjunctionPatternMatch {

  implicit class DisjunctionWithFilter[A, B](self: A \/ B) {
    def withFilter(p: B => Boolean) =
      self map { b =>
        if (p(b)) b else throw new MatchError(b)
      }
  }

}
```

```scala
import DisjunctionPatternMatch._

def printUser(id: Int): Error \/ String =
  for {
    (name, age) <- DB.loadUser(id)
  } yield s"User $name is $age years old"
```

Now it compiles! (and runs) 😄

The other example (with the unsafe list match) will throw a `MatchError`.

...which is what any other unsafe pattern match would do.

As long as we're careful not to call `filter` where it doesn't make sense.

Fix all monads

```
object MonadicPatternMatch {
  import scalaz.Monad
  import scalaz.syntax.monad._
  import scalaz.std.list._

  implicit class MonadWithFilter[M[_]: Monad, B](self: M[B]) {
    def withFilter(p: B => Boolean) =
      self.map(b => if (p(b)) b else throw new MatchError(b))
  }
}
```

```scala
import MonadicPatternMatch._

// safe match
def printUser[M[_]: Monad](id: Int) =
  for {
    (name, age) <- DB.loadUser[M](id)
  } yield s"User $name is $age years old"
```

Compiles! (and runs) 😀

```scala
// unsafe match
def sayHi[M[_]: Monad](lists: M[List[String]]) =
  for {
    List(title, name) <- lists
  } yield s"Greetings, $title $name"
```

Now all bad matches throw `MatchError`s instead of returning a neutral element.

```
> sayHi(List(List("Mark")))

[error] scala.MatchError: List(Mark) (of class scala.collectior
```

While this still isn't ideal, it's okay for many cases.

Conclusion

Scala is bad.

Conclusion

Scala is bad because it doesn't distinguish between pattern-matches that can fail, and those that can't.

Conclusion

Scala is bad because it doesn't distinguish between pattern-matches that can fail, and those that can't.

Idris does this better, as does Purescript.
Haskell is in the process of changing (back) to be better.

Conclusion

Scala is bad because it doesn't distinguish between pattern-matches that can fail, and those that can't.

Idris does this better, as does Purescript.
Haskell is in the process of changing (back) to be better.

In the meantime, implicit enhancements give us a decent workaround.

Update

There's an in-progress ticket by Greg Pfeil to fix this for Scala 2.12.x (x >= 4).

> SI-5365 Exhaustivity of extractors, guards, and unsealed traits.
> https://github.com/scala/scala/pull/5617

Thanks to Sam Halliday, Paul Phillips and Brian McKenna for providing additional info.