precision error handling in functional scala

errata

Uma Zalakain

alternative title

ApplicativeError is the wrong trait to have at the base of error handling

background; or why does this bother me

- +7 years of experience writing FP
- Most of it using FP for proof engineering
- Heavy user of dependent types
- Active in the Agda community
- Writing FP Scala for +2 years
- Very sad seeing services written in a statically typed language with a strong type system crash at runtime. I don’t want it

error handling?

with datatypes

```
enum Error[K]:
  case KeyMissing(k: K)
  case KeyExists(k: K)

trait KeyValueStore[F[_], K, V]:
  def get(k: K): F[Either[Error[K], V]]
  def del(k: K): F[Either[Error[K], Unit]]
  def put(k: K, v: V): F[Either[Error[K], Unit]]
```

lifting into an effect

```scala
enum Error[K]:
  case KeyMissing(k: K)
  case KeyExists(k: K)

trait KeyValueStore[F[_]: ApplicativeError[F, Error[K]], K, V]:
  def get(k: K): F[V]
  def del(k: K): F[Unit]
  def put(k: K, v: V): F[Unit]
```

because

ApplicativeError is the wrong trait to have at the base of error handling

in cats

```scala
trait ApplicativeError[F[_], E] extends Applicative[F]:
  def raiseError[A](e: E): F[A]
  def handleErrorWith[A](fa: F[A])(f: E => F[A]): F[A]
```

cannot distinguish between error raising and error handling

```
def method[F[_]: ApplicativeError[*, AppError],A](fa: F[A]): F[A]
```

only Throwable errors can be handled

```
case class E1()
case class E2()

def method[F[_], A](using ApplicativeError[F, E1], ApplicativeError[F, E2])(fa: F[A]): F[A] =
  Applicative[F].map(fa)(identity)
```

```
> Ambiguous given instances: both parameter x$2 and parameter x$1 match type
cats.Applicative[F] of parameter instance of method apply in object Applicative
```

```
case class E1() extends Throwable
case class E2() extends Throwable

def method[F[_], A](using ApplicativeThrow[F])(fa: F[A]): F[A] =
  fa.handleErrorWith {
    case E1() => fa
    case E2() => fa           // this is what we want
    case _: Throwable => fa // must always have this
  }
```

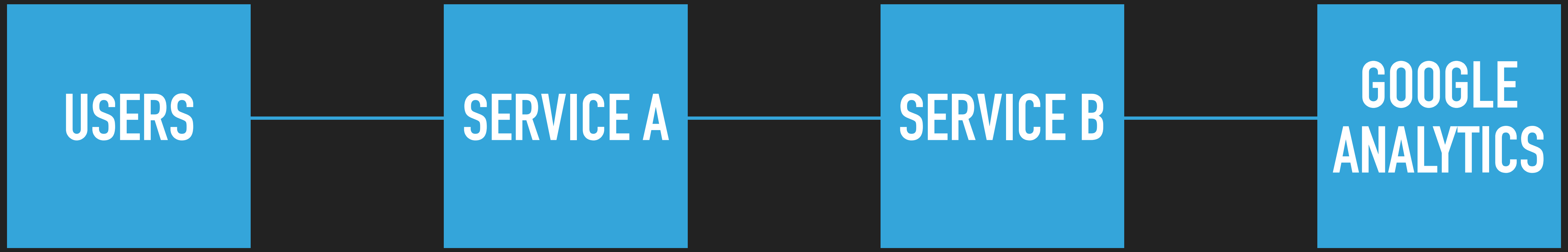
not reflected at the type level

```scala
trait ApplicativeError[F[_], E] extends Applicative[F]:

  def attempt[A](fa: F[A]): F[Either[E, A]]
```

```scala
object AppError extends Throwable

def kafkaFunc[F[_]: MonadThrow]: F[Unit] =
  MonadThrow[F].raiseError(AppError)

def dbFunc[F[_]: MonadThrow]: F[Unit] =
  MonadThrow[F].raiseError(AppError)

object App extends IOApp:
  def run(args: List[String]): IO[ExitCode] =
    for
      _ <- kafkaFunc[IO]
      _ <- dbFunc[IO]
    yield ExitCode.Success
```

```
[error] AppError$
[error]         at AppError$.<clinit>(httpClient.scala:20)
[error]         at httpClient$package$.kafkaFunc(httpClient.scala:23)
[error]         at App$.run(httpClient.scala:31)
[error]         at flatMap @ App$.run(httpClient.scala:33)
[error]         at main$ @ App$.main(httpClient.scala:28)
[error]         at main$ @ App$.main(httpClient.scala:28)
[error]         at main$ @ App$.main(httpClient.scala:28)
[error] Nonzero exit code returned from runner: 1
```

real-life consequences



more real-life consequences

$$\begin{tikzcd} F[\text{Either}[E, A]] \arrow[r] & \begin{array}{c} F[A] \\ \text{given ApplicativeThrow}[F] \end{array} \end{tikzcd}$$

this is horrible :(

errata: proof of concept, but already usable

- https://github.com/umazalakain/errata
- Based on ToFu (https://github.com/tofu-tf/tofu/)
- Solves *all* of the aforementioned
- Fully cats compatible!
- Ultimate goal: change cats instead
- You can already use it and benefit from it!

```scala
trait Raise[F[_], -E]:
  def raise[A](err: E): F[A]
```

```scala
trait HandleTo[F[_], G[_], +E]:
  def handleWith[A](fa: F[A])(f: E => G[A]): G[A]
```

```scala
trait Handle[F[_], +E]
  extends HandleTo[F, F, E]

trait ErrorsTo[F[_], G[_], E]
  extends Raise[F, E]
  with HandleTo[F, G, E]

trait TransformTo[F[_], G[_], +E1, -E2]
  extends HandleTo[F, G, E1]
  with Raise[G, E2]

trait Errors[F[_], E]
  extends Raise[F, E]
  with Handle[F, E]
  with ErrorsTo[F, F, E]
  with TransformTo[F, F, E, E]
```

~~cannot distinguish between error raising and error handling~~

```scala
final case class ClientError()
type Result

// Only raises errors
def producer[F[_]](using Raise[F, ClientError]): F[Result] =
  ClientError().raise[F]

// Only handles errors -- doesn't raise them!
def consumer[F[_]: Applicative](fp: F[Result])(using Handle[F, ClientError]): F[Unit] =
  fp.void.handleWith(_ => ().pure[F])

// Needs to do both
def process[F[_]: Applicative](using Errors[F, ClientError]): F[Unit] =
  consumer(producer)
```

~~only Throwable errors can be handled~~

```scala
// Only raises errors of type ClientError
def producerA[F[_]](using Raise[F, ClientError]): F[Result] =
  ClientError().raise[F]

// Only raises errors of type DBError
def producerB[F[_]](using Raise[F, DBError]): F[Result] =
  DBError().raise[F]

// Only handles errors, both of type ClientError and of type DBError
def consumer[F[_]: Applicative](fp: F[Result])(using Handle[F, AppError]): F[Unit] =
  fp.void.handleWith {
    // No need to handle anything outside of AppError
    case ClientError() => ().pure[F]
    case DBError() => ().pure[F]
  }

// Raises and handles errors of type AppError
def process[F[_]: Monad](using Errors[F, AppError]): F[Unit] =
  List(producerA, producerB).traverse(consumer).void
```

```scala
export AppError.*
enum AppError:
  case ClientError()
  case DBError()
type Result
```

~~not reflected at the type level~~

```scala
// Must handle all errors
def attempt[F[_]: Applicative, G[_]: Applicative, E, A](fp: F[A])(
  using HandleTo[F, G, E]
): G[Either[E, A]] =
  fp.map(_.asRight).handleWith[G, E](_.asLeft.pure)
```

other features

- Convenience syntax on effects and error types
- Common data type instances
- Interoperability with cats (in both directions)
- Property tests based on algebraic laws

taking ApplicativeError apart

ApplicativeError[F, E] = Raise[F, E] + HandleTo[F, F, E] + Applicative[F]

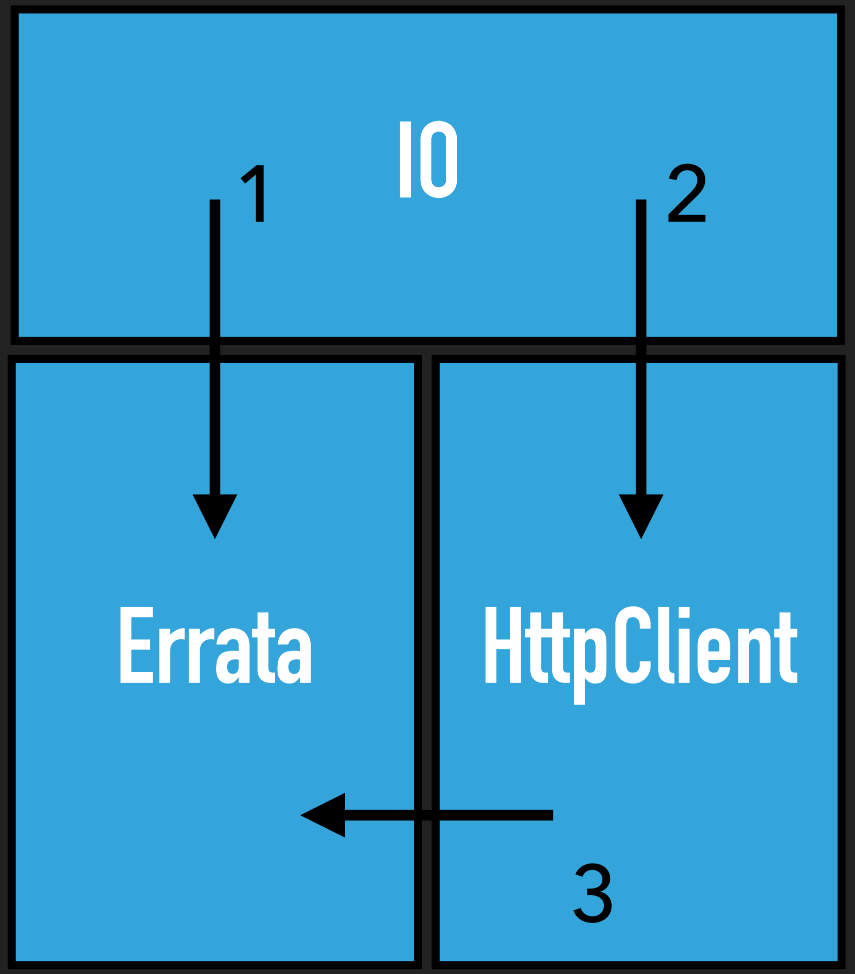
interoperability — upwards

```scala
import cats.effect.IO
import errata.*
import errata.instances.*

sealed trait AppError

implicit val appErrors: Errors[IO, AppError] =
  errorsThrowable(classTag[AppError])
```

interoperability — downwards

```scala
import cats.effect.IO
import cats.{Applicative, MonadThrow}
import errata.*
import errata.instances.*

trait HttpClient[F[_]]:
  def run[A]: F[A]

object HttpClient:
  def apply[F[_]](using MonadThrow[F]): HttpClient[F] = ???

sealed trait AppError
case class RestAPIError(th: Throwable) extends AppError

def appLogic[F[_], G[_]: Applicative, A](
    httpClient: HttpClient[F]
)(using transformTo: TransformTo[F, G, Throwable, AppError]): G[Unit] =
  httpClient.run[A].transform(RestAPIError.apply).void
```

```scala
trait HttpClient[F[_]] { def run[A]: F[A] }

object HttpClient:                                2
  def apply[F[_]](implicit F: MonadThrow[F]): HttpClient[F] =
    new HttpClient[F] {
      override def run[A]: F[A] = F.raiseError(new Throwable("Some kind of error"))

sealed trait AppError
case class RestAPIError(th: Throwable) extends AppError
case class GraphQLError(th: Throwable) extends AppError

def appLogic[F[_], G[_]: Applicative, H[_]: Console, A](httpClient: HttpClient[F])(using
 transformTo: TransformTo[F, G, Throwable, AppError],
 handleTo: HandleTo[G, H, AppError]
): H[Unit] =                                                3
  val apiResponse: G[A] = httpClient.run[A].transform(RestAPIError.apply)
  val graphqlResponse: G[A] = httpClient.run[A].transform(GraphQLError.apply)
  (apiResponse, graphqlResponse)
    .mapN { case (_, _) => () }
    .handleWith[H, AppError] {
      case RestAPIError(th) => Console[H].println(s"REST API error: ${th.getMessage}")
      case GraphQLError(th) => Console[H].println(s"GraphQL error: ${th.getMessage}")
    }

object httpClient extends IOApp:
  def run(args: List[String]): IO[ExitCode] =
    given val appErrors: Errors[IO, AppError] = errorsThrowable(classTag[AppError])
    IO.println("Expecting a properly handled error") *>                            1
      appLogic[IO, IO, IO, Unit](HttpClient[IO]).as(ExitCode.Success)
```



open questions

- What is the story with cats-effect?
- What is the story with fs2?
- What do you think or error handling in cats?
- Do you find this valuable?
- Would you use this?
- Would you like to help change things?
- What am I missing?

tl;dr

- ApplicativeError is the wrong trait to have at the base of error handling
- More precise error handling is possible
- Fixes runtime errors
- Use Errata to benefit from it today
- Let’s change cats
- I need some allies in this

- Open to work
- https://linkedin.com/in/uma-zalakain
- https://umazalakain.info
- ping@umazalakain.info