Rethinking Monad Transformers with Raise Capability

Thanh Le

20-08-2025

Disclaimer

- Credits go to the Daniel Spiewak
- Scala 3 syntax, but Scala 2 is also supported
- Focus IO (and Future) effect

About me

- Born and raised in Vietnam
- Live in Sweden and work at Recorded Future
- Love functional programming and performance optimization
- Maintainer of some open source projects, most notably lichess.org
- Chess & calisthenics

Outline

- Motivation
- Introduce our case study
- Different techniques of error handling in Scala
    - Untyped Errors
    - Typed Errors with nested Either
    - EitherT monad transformer
    - Monadic embedding of capabilities using cats-mtl
- Under the Hood (briefly)

Errors and exceptions

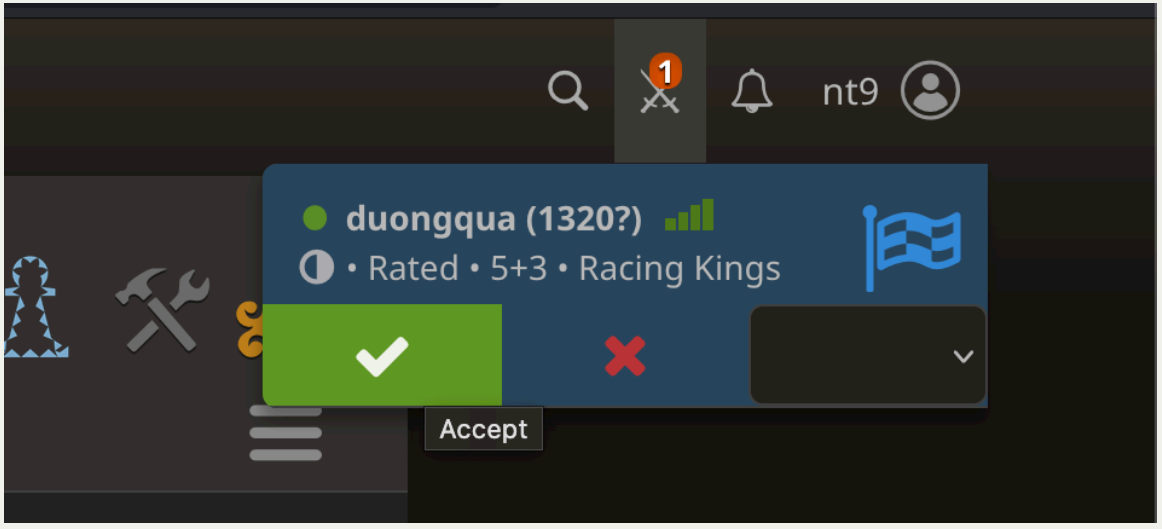
There are three kinds of errors

- Domain specific errors
- System errors
- Bugs

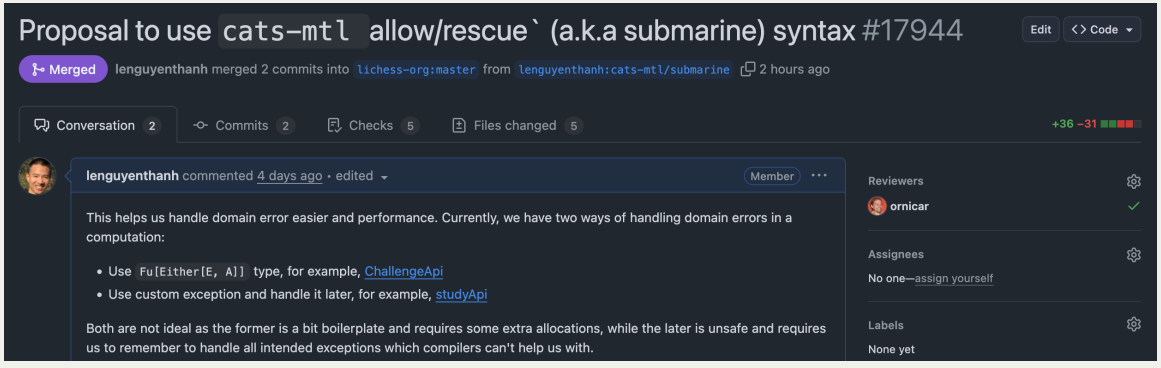
Motivation

- Safe
- Simple & concise
- Performant

Case study: accept a game challenge



Case study: accept a game challenge



Case study: accept a game challenge

```
1  def acceptChallenge(id: ChallengeId): IO[Game] =
2    for
3      challenge <- find(id)
4      _ <- accept(challenge)
5      game <- create(challenge)
6    yield game
7
8  // Challenge must exist
9  def find(id: ChallengeId)       : IO[Challenge]
10 // challenge must be active
11 def accept(challenge: Challenge): IO[Unit]
12 // Can only play one game at a time
13 def create(challenge: Challenge): IO[Game]
```

Case study: accept a game challenge

```scala
// business logic
def acceptChallenge(id: ChallengeId): IO[Game] =
  for
    challenge <- find(id)
    _ <- accept(challenge)
    game <- create(challenge)
  yield game

// Usage, for example in service/API layer
acceptChallenge("challenge", "user")
  .flatMap: game =>
    IO.println(s"Challenge accepted, game created: $game")
```

Untyped Errors

```
1  case class NotFound(id: ChallengeId)
2      extends RuntimeException(s"Challenge with id $id not found")
3  def find(id: ChallengeId): IO[Challenge]
4
5  case class IsDeclined(id: Challenge)
6      extends RuntimeException(s"Challenge $id is not for user $playerId")
7  case class IsCancelled(id: ChallengeId)
8      extends RuntimeException(s"Challenge $id is canceled")
9  def accept(challenge: Challenge): IO[Challenge]
10
11 case class CreateGameError(message: String)
12     extends RuntimeException(message)
13 def create(challenge: Challenge): IO[Game]
```

Untyped Errors

```
1  acceptChallenge("challenge", "user")
2    .flatMap: game =>
3      IO.println(s"Challenge accepted, game created: $game")
4
5  acceptChallenge("challenge", "user")
6    .flatMap: game =>
7      IO.println(s"Challenge accepted, game created: $game")
8    .recoverWith:
9      case NotFound(id) => IO.unit
10     case IsDeclined(challenge) => IO.unit
11     case CreateGameError(message) => IO.unit
```

Untyped Errors

```
1 acceptChallenge("challenge", "user")
2   .flatMap: game =>
3     IO.println(s"Challenge accepted, game created: $game")
4   .recoverWith:
5     case NotFound(id) => IO.unit
6     case IsDeclined(challenge) => IO.unit
7     case IsCancelled(id) => IO.unit
8     case CreateGameError(message) => IO.unit
```

Untyped Errors

```
1 acceptChallenge("challenge", "user")
2   .flatMap: game =>
3     IO.println(s"Challenge accepted, game created: $game")
4   .recoverWith:
5     case NotFound(id) => IO.unit
6     case IsDeclined(challenge) => IO.unit
7     case IsCancelled(id) => IO.unit
8     case CreateGameError(message) => IO.unit
9     case _ => IO.unit // catch all other exceptions
```

Untyped Errors

- There is no exhaustive check for pattern matching
- We cannot distinguish between different kinds of errors
- Exceptions are hidden from the function signature

How do we know what exceptions can be thrown?

- Read documentation
- Read the implementation and figure it out.
- Run the code and see what it actually throws (or our users)

Conclusion for Untyped Errors

- ~~Safe~~
- ~~Simple and concise~~
- Performant

Typed errors with nested Either

```scala
// Look Ma, no more RuntimeException here!
case class NotFound(id: ChallengeId)

enum AcceptError:
  case IsDeclined(challenge: Challenge)
  case IsCanceled(id: ChallengeId)

case class CreateGameError(message: String)

type Error = NotFound | AcceptError | CreateGameError
```

Typed errors with nested Either

```scala
def find(id: ChallengeId):         IO[Either[NotFound, Challenge]]
def accept(challenge: Challenge): IO[Either[AcceptError, Unit]]
def create(challenge: Challenge): IO[Either[CreateGameError, Game]]

def acceptChallenge(id: ChallengeId): IO[Either[Error, Game]] =
  for
    challenge <- find(id)
    result <- accept(challenge)
    game <- create(result)
  yield game

acceptChallenge("challenge", "user")
  .flatMap:
    case Right(game) =>
      IO.println(s"Challenge accepted, game created: $game")
    case Left(NotFound(id)) => IO.unit
    case Left(IsDeclined(challenge)) => IO.unit
    case Left(IsCanceled(id)) => IO.unit
    case Left(CreateGameError(message)) => IO.unit
```

this does not compile :cry:

```scala
// We can't compose many IO[Either[A, B]] together
def acceptChallenge(id: ChallengeId): IO[Either[Error, Game]] =
  for
    challenge <- find(id)
    result <- accept(challenge)
    game <- create(result)
  yield game
```

Let's fix it

```scala
def acceptChallenge(id: ChallengeId): IO[Either[Error, Game]] =
  for
    challengeOrError <- find(id)
    acceptedOrError <- challengeOrError match
      case Left(error) => IO.pure(Left[Error, Unit](error))
      case Right(challenge) => accept(challenge).map(_.as(challenge))
    game <- acceptedOrError match
      case Left(error) => IO.pure(Left[Error, Game](error))
      case Right(challenge) => create(challenge)
  yield game
```

Or we can do it with nested flatMap

```
1  def acceptChallenge(id: ChallengeId): IO[Either[Error, Game]] =
2      find(id).flatMap:
3        case Left(error) => IO.pure(Left(error))
4        case Right(challenge) =>
5          accept(challenge).flatMap:
6            case Left(error) => IO.pure(Left(error))
7            case Right(_) =>
8              create(challenge).map:
9                case Left(error) => Left(error)
10               case Right(game) => Right(game)
```

Conclusion for typed error with nested Either

- Safe
- ~~Simple and concise~~
- Performant

EitherT (monad transformers)

```scala
import cats.data.EitherT
def accept(id: ChallengeId): IO[Either[Error, Game]] =
  val eitherT: EitherT[IO, Error, Game] =
    for
      challenge <- EitherT(find(id))
      _ <- EitherT(accept(challenge))
      game <- EitherT(create(challenge))
    yield game
  eitherT.value
```

Few words about EitherT

```
// EitherT.apply(IO[Either[A, B]]):  EitherT[IO, A, B]
// EitherT[IO, A, B].value: IO[Either[A, B]
class EitherT[F[_], A, B] private (val value: F[Either[A, B]]):
  def flatMap[C](f: B => EitherT[F, A, C]): EitherT[F, A, C] = ???
  def map[C](f: B => C): EitherT[F, A, C] = ???
object EitherT:
  def apply[F[_], A, B](value: F[Either[A, B]]): EitherT[F, A, B] =
    new EitherT(value)
```

Conclusion for EitherT

- Safe
- ~~Simple and concise~~
- ~~Performant~~

Notes on EitherT and cats-effect

- There are some limitation with concurrent code based
  - https://github.com/typelevel/fs2/issues/319
  - https://github.com/typelevel/cats/issues/43
  - https://github.com/typelevel/cats-effect/discussions/3765
  - https://github.com/typelevel/fs2/pull/2895
  - https://github.com/typelevel/cats-effect/issues/2448

cats-mtl with Raise capability

```scala
//> using dep org.typelevel::cats-mtl:1.6.0
import cats.mtl.Raise

def find(id: ChallengeId)(using Raise[IO, NotFound]): IO[Challenge]
def accept(challenge: Challenge)(using Raise[IO, AcceptError]): IO[Unit]
def create(challenge: Challenge)(using Raise[IO, CreateGameError]): IO[Game]

// type Error = NotFound | AcceptError | CreateGameError
def acceptChallenge(id: ChallengeId)(using Raise[IO, Error]): IO[Game] =
  for
    challenge <- find(id)
    _ <- accept(challenge)
    game <- create(challenge)
  yield game
```

cats-mtl with Raise capability

```
1 import cats.mtl.Handle
2 Handle.allow[Error]:
3   acceptChallenge("challenge", "user").flatMap: game =>
4     IO.println(s"Challenge accepted, game created: $game")
5 .rescue:
6   case NotFound(id) => IO.unit
7   case IsDeclined(challenge) => IO.unit
8   case IsCanceled(id) => IO.unit
9   case CreateGameError(message) => IO.unit
```

cats-mtl with Raise capability

```scala
Handle.allow[Error]:
  acceptChallenge("challenge", "user").flatMap: game =>
    IO.println(s"Challenge accepted, game created: $game")
.rescue:
  case NotFound(id) => IO.unit
  case IsDeclined(challenge) => IO.unit
  case IsCanceled(id) => IO.unit
  case CreateGameError(message) => IO.unit
```

cats-mtl with Raise capability

```scala
// context function
type IORaise[E, A] = Raise[IO, E] ?=> IO[A]

def find(id: ChallengeId):         IORaise[NotFound, Challenge]
def accept(challenge: Challenge): IORaise[AcceptChallengeError, Unit]
def create(challenge: Challenge): IORaise[CreateGameError, Game]

def acceptChallenge(id: ChallengeId): IORaise[Error, Game] =
  for
    challenge <- findChallenge(id)
    _ <- accept(challenge)
    game <- createGame(challenge)
  yield game
```

cats-mtl with Raise capability

```scala
// scala 2
allowF[IO, Error] { implicit h =>
  acceptChallenge("challenge", "user").flatMap{ game =>
    IO.println(s"Challenge accepted, game created: $game")
 }
}.rescue {
  case NotFound(id) => IO.unit
  case IsDeclined(challenge) => IO.unit
  case IsCanceled(id) => IO.unit
  case CreateGameError(message) => IO.unit
}
```

Conclusion for cats-mtl

- Safe
- Simple and concise
- Performant

Some caveats

- Additional dependency on `cats-mtl`
- a bit of new concepts to learn (e.g. `Raise`, `allow`, `rescue`)
- Compilation error messages can be cryptic

Under the hood

- Re-use existing abstraction from cats and cats-mtl
- Context functions (the A ?=> B syntax)
- inline functions

links

- The cats-mtl pr: https://github.com/typelevel/cats-mtl/pull/619
- PRs of using cats-mtl in lichess
  - https://github.com/lichess-org/lila-search/pull/542
  - https://github.com/lichess-org/lila/pull/17944

thank you