

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

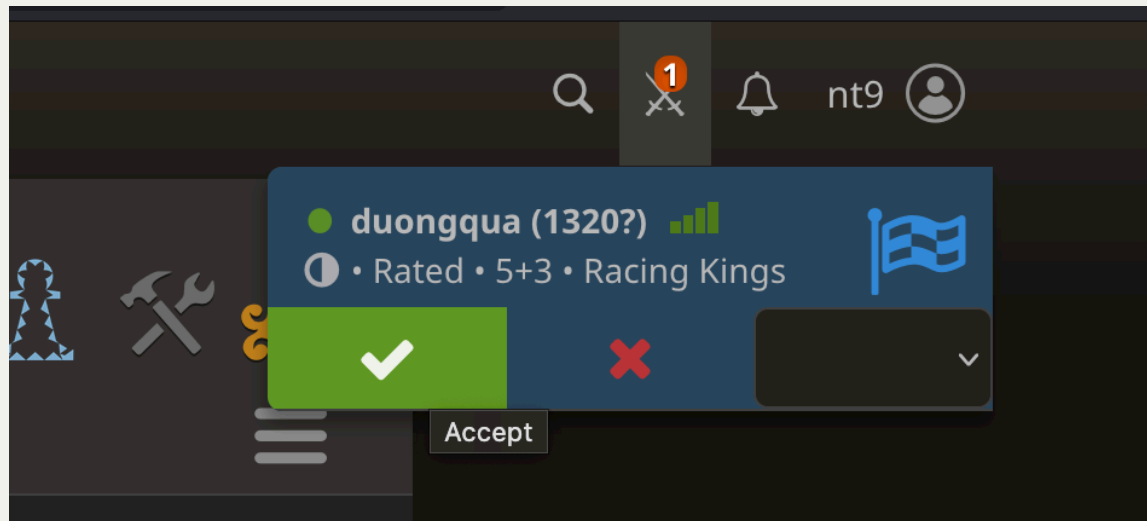
There are three kinds of errors

- Domain specific errors
- System errors
- Bugs

Motivation

- Safe
- Simple & concise
- Performant

Case study: accept a game challenge



Case study: accept a game challenge

The screenshot shows a GitHub pull request interface. At the top, the title is "Proposal to use `cats-mtl allow/rescue`` (a.k.a submarine) syntax #17944". Below the title, it says "Merged" and "lenguyenthanh merged 2 commits into lichess-org:master from lenguyenthanh:cats-mtl/submarine 2 hours ago".

Below the merge information, there are tabs for "Conversation 2", "Commits 2", "Checks 5", and "Files changed 5". To the right of these tabs, there is a green bar indicating "+36 -31" changes.

The main content area shows a comment by "lenguyenthanh" (Member) from 4 days ago. The comment text is: "This helps us handle domain error easier and performance. Currently, we have two ways of handling domain errors in a computation:".

- Use `Fu[Either[E, A]]` type, for example, [ChallengeApi](#)
- Use custom exception and handle it later, for example, [studyApi](#)

Below the list, the comment continues: "Both are not ideal as the former is a bit boilerplate and requires some extra allocations, while the later is unsafe and requires us to remember to handle all intended exceptions which compilers can't help us with."

On the right side of the comment, there are sections for "Reviewers", "Assignees", and "Labels". Under "Reviewers", "ornicar" is listed with a green checkmark. Under "Assignees", it says "No one—assign yourself". Under "Labels", it says "None yet".

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

```
1 // business logic
2 def acceptChallenge(id: ChallengeId): IO[Game] =
3   for
4     challenge <- find(id)
5     _ <- accept(challenge)
6     game <- create(challenge)
7   yield game
8
9 // Usage, for example in service/API layer
10 acceptChallenge("challenge", "user")
11   .flatMap: game =>
12     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

```
1 // Look Ma, no more RuntimeException here!
2 case class NotFound(id: ChallengeId)
3
4 enum AcceptError:
5   case IsDeclined(challenge: Challenge)
6   case IsCanceled(id: ChallengeId)
7
8 case class CreateGameError(message: String)
9
10 type Error = NotFound | AcceptError | CreateGameError
```

Typed errors with nested Either

```
1 def find(id: ChallengeId): IO[Either[NotFound, Challenge]]
2 def accept(challenge: Challenge): IO[Either[AcceptError, Unit]]
3 def create(challenge: Challenge): IO[Either[CreateGameError, Game]]
4
5 def acceptChallenge(id: ChallengeId): IO[Either[Error, Game]] =
6   for
7     challenge <- find(id)
8     result <- accept(challenge)
9     game <- create(result)
10  yield game
11
12 acceptChallenge("challenge", "user")
13   .flatMap:
14     case Right(game) =>
15       IO.println(s"Challenge accepted, game created: $game")
16     case Left(NotFound(id)) => IO.unit
17     case Left(IsDeclined(challenge)) => IO.unit
18     case Left(IsCanceled(id)) => IO.unit
19     case Left(CreateGameError(message)) => IO.unit
```

this does not compile :cry:

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

Let's fix it

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

```
1 import cats.data.EitherT
2 def accept(id: ChallengeId): IO[Either[Error, Game]] =
3   val eitherT: EitherT[IO, Error, Game] =
4     for
5       challenge <- EitherT(find(id))
6       _ <- EitherT(accept(challenge))
7       game <- EitherT(create(challenge))
8     yield game
9   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

```
1 //> using dep org.typelevel::cats-mtl:1.6.0
2 import cats.mtl.Raise
3
4 def find(id: ChallengeId)(using Raise[IO, NotFound]): IO[Challenge]
5 def accept(challenge: Challenge)(using Raise[IO, AcceptError]): IO[Unit]
6 def create(challenge: Challenge)(using Raise[IO, CreateGameError]): IO[Game]
7
8 // type Error = NotFound | AcceptError | CreateGameError
9 def acceptChallenge(id: ChallengeId)(using Raise[IO, Error]): IO[Game] =
10   for
11     challenge <- find(id)
12     _ <- accept(challenge)
13     game <- create(challenge)
14   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

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

cats-mtl with Raise capability

```
1 // context function
2 type IORaise[E, A] = Raise[IO, E] ?=> IO[A]
3
4 def find(id: ChallengeId): IORaise[NotFound, Challenge]
5 def accept(challenge: Challenge): IORaise[AcceptChallengeError, Unit]
6 def create(challenge: Challenge): IORaise[CreateGameError, Game]
7
8 def acceptChallenge(id: ChallengeId): IORaise[Error, Game] =
9   for
10     challenge <- findChallenge(id)
11     _ <- accept(challenge)
12     game <- createGame(challenge)
13   yield game
```

cats-mtl with Raise capability

```
1 // scala 2
2 allowF[IO, Error] { implicit h =>
3   acceptChallenge("challenge", "user").flatMap{ game =>
4     IO.println(s"Challenge accepted, game created: $game")
5   }
6 }.rescue {
7   case NotFound(id) => IO.unit
8   case IsDeclined(challenge) => IO.unit
9   case IsCanceled(id) => IO.unit
10  case CreateGameError(message) => IO.unit
11 }
```


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 $A \Rightarrow B$ 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