An interpreter handling over effects for Eff

Yoshimura Hikaru（吉村 優）

hikaru_yoshimura@r.recruit.co.jp

Recruit Marketing Partners Co., Ltd.

October 17, 2020 @ ScalaMatsuri 2020

https://github.com/y-yu/scalamatsuri2020 (e6fc40b)

Table of contents

Who am I?

Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

Who am I?

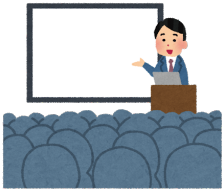
- Recruit Marketing Partners Co., Ltd.
  - StudySapuri ENGLISH server side(Scala)
- Quantum Information
  - but I don't know well about Quantum annealing...
- Cryptography & Security
- LaTeX typesetting

Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

Concrete case I'll talk about

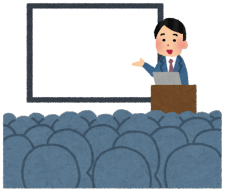
Concrete case I'll talk about

In this talk, we think about one concrete case:



Concrete case I'll talk about

In this talk, we think about one concrete case:



“ *Read & Write data to database with the transaction* ”

- This is very common case in the programming, but there are many ways to do it

Low level example

Low level example

```
val transactionManager = new TM()
val session = transactionManager.begin()

databaseOperation(
  // If you want to rollback,
  // call `session.fail`
  session
)

if (transactionManager.commit(session))
  /* Successful */
else
  /* Failure */
```

Low level example

```
val transactionManager = new TM()
val session = transactionManager.begin()

databaseOperation(
  // If you want to rollback,
  // call `session.fail`
  session
)

if (transactionManager.commit(session))
  /* Successful */
else
  /* Failure */
```

- It's (maybe) used in tranditional languages like C

Low level example

```
val transactionManager = new TM()
val session = transactionManager.begin()

databaseOperation(
  // If you want to rollback,
  // call `session.fail`
  session
)

if (transactionManager.commit(session))
  /* Successful */
else
  /* Failure */
```

- It's (maybe) used in tranditional languages like C
- You know, that way has some problems:

Low level example

```
val transactionManager = new TM()
val session = transactionManager.begin()

databaseOperation(
  // If you want to rollback,
  // call `session.fail`
  session
)

if (transactionManager.commit(session))
  /* Successful */
else
  /* Failure */
```

- It's (maybe) used in tranditional languages like C
- You know, that way has some problems:

Could programmers *forget* to write `begin` and `commit`?

Loan pattern

- If the problem is forgetting, we can use *Loan pattern*:

```
def withTransaction(
  f: Session => A
): Either[Throwable, A] = {
  val session = transactionManager.begin()
  val a = f(session)
  if (transactionManager.commit(session)) Right(a)
  else Left(new RuntimeException())
}

withTransaction { session =>
  something.databaseOperation(session)
}
```

Loan pattern

- If the problem is forgetting, we can use *Loan pattern*:

```
def withTransaction(
  f: Session => A
): Either[Throwable, A] = {
  val session = transactionManager.begin()
  val a = f(session)
  if (transactionManager.commit(session)) Right(a)
  else Left(new RuntimeException())
}

withTransaction { session =>
  something.databaseOperation(session)
}
```

- `withTransaction` takes a function `f`

Loan pattern

- If the problem is forgetting, we can use *Loan pattern*:

```
def withTransaction(
  f: Session => A
): Either[Throwable, A] = {
  val session = transactionManager.begin()
  val a = f(session)
  if (transactionManager.commit(session)) Right(a)
  else Left(new RuntimeException())
}

withTransaction { session =>
  something.databaseOperation(session)
}
```

- `withTransaction` takes a function `f`
- And then execute it inside the `begin` and `commit`

Loan pattern

- If the problem is forgetting, we can use *Loan pattern*:

```
def withTransaction(
  f: Session => A
): Either[Throwable, A] = {
  val session = transactionManager.begin()
  val a = f(session)
  if (transactionManager.commit(session)) Right(a)
  else Left(new RuntimeException())
}

withTransaction { session =>
  something.databaseOperation(session)
}
```

- `withTransaction` takes a function `f`
- And then execute it inside the `begin` and `commit`

Is Loan pattern the silver bullet?

Nested loan pattern

Nested loan pattern

- We can use `withTransaction` *illegally* like below

```
def ops(): Either[Throwable, ?] =
  withTransaction { session =>
    something.databaseOperation(session)
  }

withTransaction { session =>
  /* something using session */
  ops()
}
```

Use `withTransaction` in the other `withTransaction` 😈

Nested loan pattern

- We can use `withTransaction` *illegally* like below

```
def ops(): Either[Throwable, ?] =
  withTransaction { session =>
    something.databaseOperation(session)
  }

withTransaction { session =>
  /* something using session */
  ops()
}
```

Use `withTransaction` in the other `withTransaction` 😈

- No one wants to do that but it's allowed... 😇

Nested loan pattern

- We can use `withTransaction` *illegally* like below

```
def ops(): Either[Throwable, ?] =
  withTransaction { session =>
    something.databaseOperation(session)
  }

withTransaction { session =>
  /* something using session */
  ops()
}
```

Use `withTransaction` in the other `withTransaction` 😈

- No one wants to do that but it's allowed... 😇
- Indeed, we don't actually "forget" to write `begin` and `commit`, but the other problem remains
  - In addtion, the first low level example **also** has this problem

Monad

Monad

- We can use `map` and `flatMap` instead of raw Loan pattern

Monad

- We can use `map` and `flatMap` instead of raw Loan pattern

```
case class DBIO[A](
  run: Session => A
) {
  def map[B](f: A => B): DBIO[B] =
    flatMap(a => DBIO(_ => f(a)))

  def flatMap[B](f: A => DBIO[B]): DBIO[B] =
    DBIO(s => f(run(s)).run(s))
}
```

Monad

- We can use `map` and `flatMap` instead of raw Loan pattern

```
case class DBIO[A](
  run: Session => A
) {
  def map[B](f: A => B): DBIO[B] =
    flatMap(a => DBIO(_ => f(a)))

  def flatMap[B](f: A => DBIO[B]): DBIO[B] =
    DBIO(s => f(run(s)).run(s))
}
```

- And define this utility function: `ask`

```
object DBIO {
  def ask: DBIO[Session] =
    DBIO(s => s)
}
```

Monad

- We can use `map` and `flatMap` instead of raw Loan pattern

```
case class DBIO[A](
  run: Session => A
) {
  def map[B](f: A => B): DBIO[B] =
    flatMap(a => DBIO(_ => f(a)))

  def flatMap[B](f: A => DBIO[B]): DBIO[B] =
    DBIO(s => f(run(s)).run(s))
}
```

- And define this utility function: `ask`

```
object DBIO {
  def ask: DBIO[Session] =
    DBIO(s => s)
}
```

- We can implement code that access to the database with `DBIO`

```
def greatDBOps1: DBIO[?] =
  DBIO.ask map { session: Session =>
    session.execute(/* Great Operation! */)
  }
```

Monad

- `greatDBOps1`, `greatDBOps2` and `greatDBOps3` run in the same transaction 😆

```
val dbio: DBIO[Int] = for {
  a <- greatDBOps1
  b <- greatDBOps2
  c <- greatDBOps3(a, b)
} yield c

withTransaction { session =>
  dbio.run(session)
}
```

Monad

- `greatDBOps1`, `greatDBOps2` and `greatDBOps3` run in the same transaction 😆

Is there *well-known monad* which can do the same things?

```
val dbio: DBIO[Int] = for {
  a <- greatDBOps1
  b <- greatDBOps2
  c <- greatDBOps3(a, b)
} yield c

withTransaction { session =>
  dbio.run(session)
}
```

Monad

- `greatDBOps1`, `greatDBOps2` and `greatDBOps3` run in the same transaction 😆

Is there *well-known monad* which can do the same things?

```
val dbio: DBIO[Int] = for {
  a <- greatDBOps1
  b <- greatDBOps2
  c <- greatDBOps3(a, b)
} yield c

withTransaction { session =>
  dbio.run(session)
}
```

- Yes, `DBIO[A]` is the same as `Reader[Session, A]`

Next step

Next step

- `DBIO` represents *just* a Database I/O
  - but we sometimes want to use other (side) effects...

Next step

- `DBIO` represents *just* a Database I/O
  - but we sometimes want to use other (side) effects...

What are the other side effects?

Next step

- `DBIO` represents *just* a Database I/O
  - but we sometimes want to use other (side) effects...

What are the other side effects?

- It's time to go to the next step:

> *We want to send e-mails only if the database transaction is successful*

Email

Email

- Sending e-mail is as popular as using database

Email

- Sending e-mail is as popular as using database
- Database has transactions but e-mail *doesn't*

Email

- Sending e-mail is as popular as using database
- Database has transactions but e-mail *doesn't*
- So we want to send e-mail after all operations are done successfully

Naive approach

Naive approach

- There is a sending e-mail function that has such an interface:

```
def sendMail(
  mail: Mail
): Either[Throwable, Unit]
```

  - `Mail` consists of to-address, from-address, title and email body

Naive approach

- There is a sending e-mail function that has such an interface:

```
def sendMail(
  mail: Mail
): Either[Throwable, Unit]
```

  - `Mail` consists of to-address, from-address, title and email body

- And then we use this after the database transaction

```
val result = withTransaction { session =>
  dbio.run(session)
}

if (result.isRight)
  sendMail(greatEmail) match {
    case Left(e) => /* something */
    case _       => ()
  }
```

Naive approach

- There is a sending e-mail function that has such an interface:

```
def sendMail(
  mail: Mail
): Either[Throwable, Unit]
```

  - `Mail` consists of to-address, from-address, title and email body

- And then we use this after the database transaction

```
val result = withTransaction { session =>
  dbio.run(session)
}

if (result.isRight)
  sendMail(greatEmail) match {
    case Left(e) => /* something */
    case _       => ()
  }
```

The *code distance* between `sendMail` and DB operation is too far away

Cohesion

What is the code distance?

It's known as *cohesion*

Cohesion

What is the code distance?

It's known as *cohesion*

- We implement the DB operating function that returns `DBIO`

```
def userUpdate(newUserInfo: UserInfo): DBIO[Unit] =
  DBIO.ask map { session =>
    /* Great user update logic is here!!!! */
  }
```

Cohesion

What is the code distance?

It's known as *cohesion*

- We implement the DB operating function that returns `DBIO`

```
def userUpdate(newUserInfo: UserInfo): DBIO[Unit] =
  DBIO.ask map { session =>
    /* Great user update logic is here!!!! */
  }
```

We want to write sending e-mail logic here too!

Cohesion

What is the code distance?

It's known as *cohesion*

- We implement the DB operating function that returns `DBIO`

```
def userUpdate(newUserInfo: UserInfo): DBIO[Unit] =
  DBIO.ask map { session =>
    /* Great user update logic is here!!!! */
  }
```

We want to write sending e-mail logic here too!

- But actually we can only write e-mail logic behind the `withTransaction` 😇
  - It means that our code is low cohesion

Cohesion

OK. How about to return `DBIO[Either[Throwable, A]]`?

Cohesion

OK. How about to return `DBIO[Either[Throwable, A]]`?

```
def userUpdate(
  newUserInfo: UserInfo
): DBIO[Either[Throwable, Unit]] =
  DBIO.ask map { session =>
    val result = /* Great user update logic */
    val mail = /* Great e-mail from newUserInfo */

    if (result)
      sendMail(greatEmail)
    else
      Left(/* error! */)
  }
```

Cohesion

OK. How about to return `DBIO[Either[Throwable, A]]`?

```
def userUpdate(
  newUserInfo: UserInfo
): DBIO[Either[Throwable, Unit]] =
  DBIO.ask map { session =>
    val result = /* Great user update logic */
    val mail = /* Great e-mail from newUserInfo */

    if (result)
      sendMail(greatEmail)
    else
      Left(/* error! */)
  }
```

- Is it OK?🤔

Cohesion

OK. How about to return `DBIO[Either[Throwable, A]]`?

```
def userUpdate(
  newUserInfo: UserInfo
): DBIO[Either[Throwable, Unit]] =
  DBIO.ask map { session =>
    val result = /* Great user update logic */
    val mail = /* Great e-mail from newUserInfo */

    if (result)
      sendMail(greatEmail)
    else
      Left(/* error! */)
  }
```

- Is it OK?🤔
- This code appears to have high cohesion, unlike before

Monad transformer

We can use *monad transformer* such as `EitherT`

Monad transformer

We can use *monad transformer* such as `EitherT`

- Monad transformer takes a monadic type constructor and turns it into a monad

```
def userUpdateT(
  newUserInfo: UserInfo
): EitherT[DBIO, Throwable, Unit] =
  userUpdate(newUserInfo).toEitherT
```

- Scala's `for` only can access the most outer monad
- So if we use `EitherT` rather than `Either`, it will be easy to access `Either` monad inside `DBIO`

Email failure example

Email failure example

- You maybe know, both `DBIO[Either[Throwable, A]]` and `EitherT[DBIO, Throwable, A]` have such a problem:

The e-mail in `userUpdate` will be sent even if `maybeFail` fails

```
val dbio = for {
  // With sending e-mail here
  _ <- userUpdate(newUserInfo)
  _ <- maybeFail // 💣 😈
} yield ???

withTransaction(dbio.run)
```

Email failure example

- You maybe know, both `DBIO[Either[Throwable, A]]` and `EitherT[DBIO, Throwable, A]` have such a problem:

The e-mail in `userUpdate` will be sent even if `maybeFail` fails

```
val dbio = for {
  // With sending e-mail here
  _ <- userUpdate(newUserInfo)
  _ <- maybeFail // 💣 😈
} yield ???

withTransaction(dbio.run)
```

It's no good that database I/O are rollbacked however e-mail has been sent!

Summary up to this point

Summary up to this point

- We want **both** *cohesion and consistency*

Summary up to this point

- We want **both** *cohesion and consistency*
- Up to this point of my talk, there seems to be a trade-off between the two

Summary up to this point

- We want **both** *cohesion and consistency*
- Up to this point of my talk, there seems to be a trade-off between the two
- In my opinion, there are two ways to combine them:
  1. Make an original moand to do it
  2. Use `Eff` and implement its suitable *interpreter* for the trade-off

Summary up to this point

- We want **both** *cohesion and consistency*
- Up to this point of my talk, there seems to be a trade-off between the two
- In my opinion, there are two ways to combine them:
  1. Make an original moand to do it
  2. Use `Eff` and implement its suitable *interpreter* for the trade-off
- First, I will describe option 2. Then I will present my opinion on which is the better choice

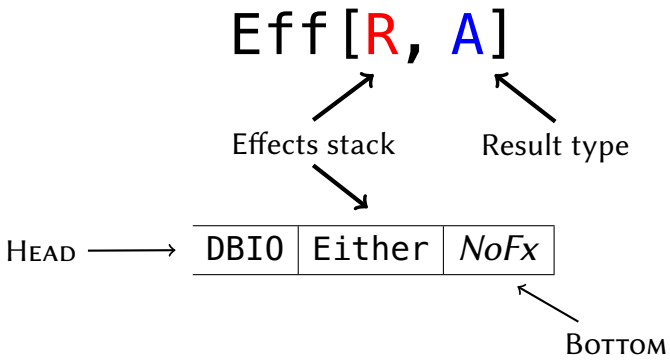
Table of contents

What is Eff*

*In this talk, `Eff` is based on `atnos-eff`.

What is Eff*

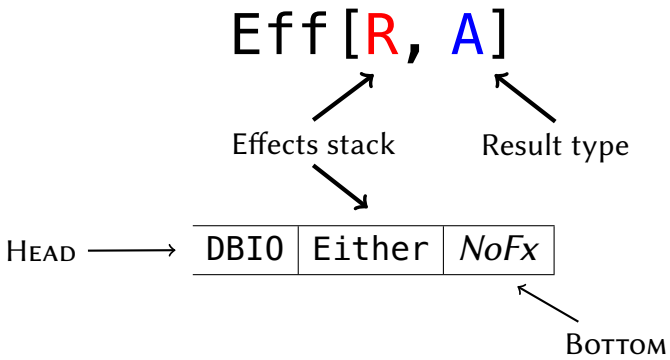
- `Eff` is a type constructor which takes two types: `R` and `A`



*In this talk, `Eff` is based on `atnos-eff`.

What is Eff*

- `Eff` is a type constructor which takes two types: `R` and `A`



- To simplify in this talk, *effects stack* is a type level stack like this ☝
  - In this figure, the effects stack has `DBIO` and `Either`

*In this talk, `Eff` is based on `atnos-eff`.

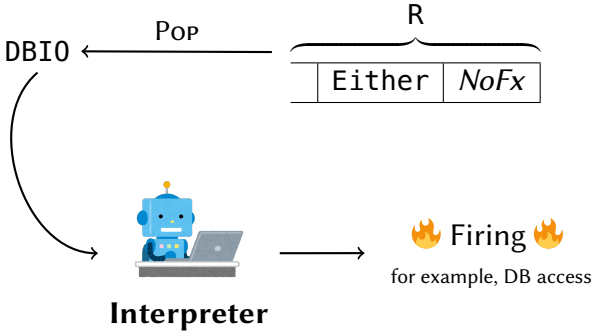
Interpreter

Interpreter

- We need *interpreters* to fire real effects

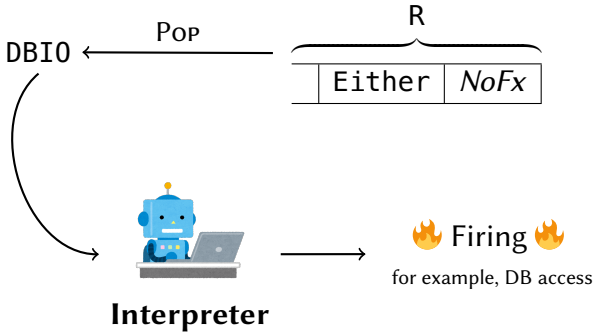
Interpreter

- We need *interpreters* to fire real effects
- When run an *interpreter*, it takes type(s) from R and execute real effects



Interpreter

- We need *interpreters* to fire real effects
- When run an *interpreter*, it takes type(s) from R and execute real effects



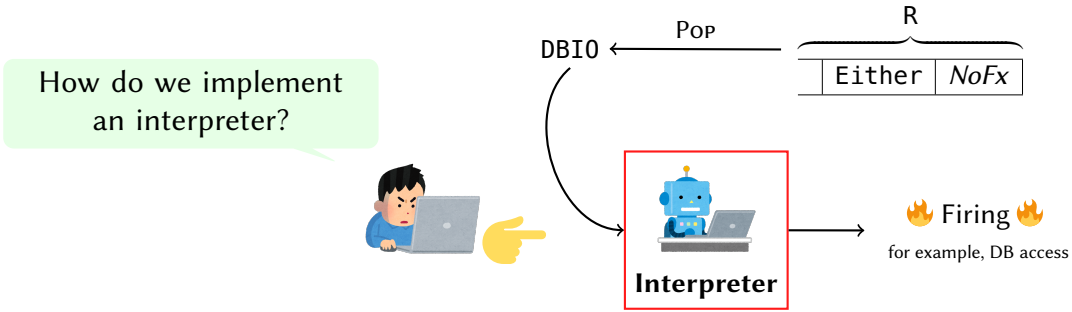
- It means that types in R are just “symbols” so they don’t have the logic for real effects
  - Firing effect logics are given by interpreters

Intuition for Eff and DI

- That's similar to *dependecy injection(DI)*, I think 🤔
  - DI Interface ← Implementation
  - Eff Type in effects stack ← Interpreter

Intuition for Eff and DI

- That's similar to *dependecy injection(DI)*, I think 🤔
  - DI Interface ← Implementation
  - Eff Type in effects stack ← Interpreter
- And then



Interpreter's interface (atnos-eff)

- This is interface of atons-eff `Interpreter`

```
trait Interpreter[M[_], R, A, B] {
  def onPure(a: A): Eff[R, B]

  def onEffect[X](x: M[X], continuation: Continuation[R, X, B]): Eff[R, B]

  def onLastEffect[X](x: M[X], continuation: Continuation[R, X, Unit]): Eff[R, Unit]

  def onApplicativeEffect[X, T[_] : Traverse](
    xs: T[M[X]], continuation: Continuation[R, T[X], B]
  ): Eff[R, B]
}
```

  - https://github.com/atnos-org/eff/blob/master/shared/src/main/scala/org/atnos/eff/Interpret.scala

Interpreter's interface (atnos-eff)

- This is interface of atons-eff `Interpreter`

```
trait Interpreter[M[_], R, A, B] {
  def onPure(a: A): Eff[R, B]

  def onEffect[X](x: M[X], continuation: Continuation[R, X, B]): Eff[R, B]

  def onLastEffect[X](x: M[X], continuation: Continuation[R, X, Unit]): Eff[R, Unit]

  def onApplicativeEffect[X, T[_] : Traverse](
    xs: T[M[X]], continuation: Continuation[R, T[X], B]
  ): Eff[R, B]
}
```

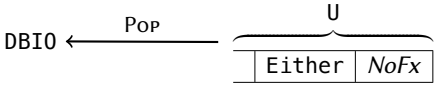
  - https://github.com/atnos-org/eff/blob/master/shared/src/main/scala/org/atnos/eff/Interpret.scala

What does it mean?

Interpreter

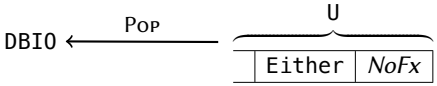
Interpreter

- We think that we run an interpreter for `DBIO`, to `R` that is $\boxed{\texttt{DBIO} \mid \texttt{Either} \mid \textit{NoFx}}$ of `Eff[R, A]`



Interpreter

- We think that we run an interpreter for `DBIO`, to `R` that is | `DBIO` | `Either` | *NoFx* | of `Eff[R, A]`

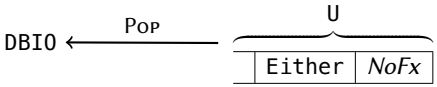


- An interpreter provides two values for us:
  1. `DBIO[X]`
  2. continuation: `X => Eff[U, B]`

  to implement the monad instace for the effect
  - So we define `map` and `flatMap` from the two parts

Interpreter

- We think that we run an interpreter for `DBIO`, to `R` that is | `DBIO` | `Either` | *NoFx* | of `Eff[R, A]`



- An interpreter provides two values for us:
  1. `DBIO[X]`
  2. continuation: `X => Eff[U, B]`

  to implement the monad instace for the effect
  - So we define `map` and `flatMap` from the two parts

What is continuation?

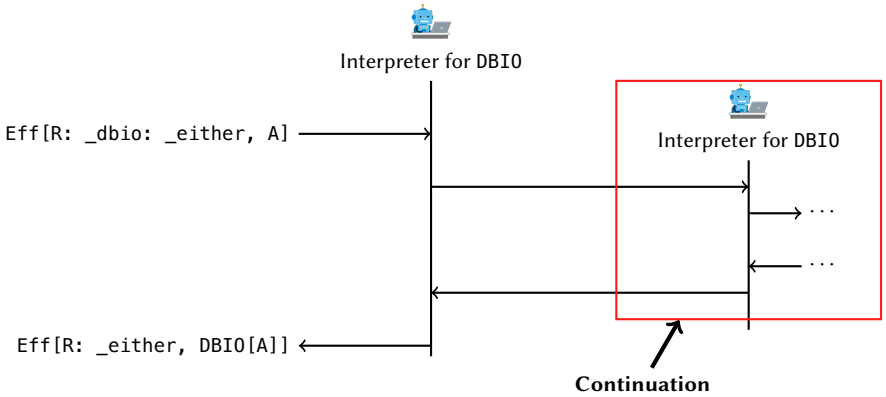
Notation

- First, we introduce a new notation to R before explain
- Assuming that `R: _dbio: _either`, it means R is

`DBIO`	`Either`	*NoFx*

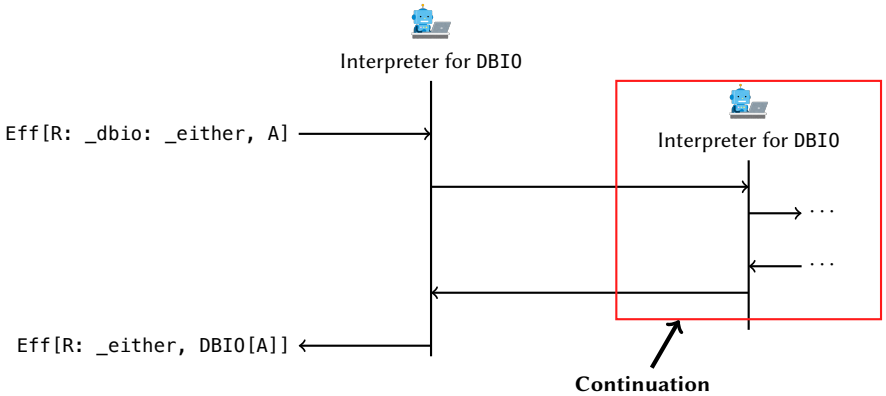
Continuation for interpreter

- There are some `DBIO` operations in the `Eff[R: _dbio: _either, A]`



Continuation for interpreter

- There are some `DBIO` operations in the `Eff[R: _dbio: _either, A]`

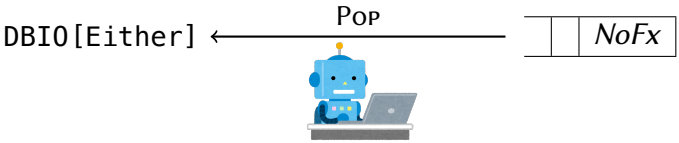


- First interpreter can access the continuation as a function, which processes effect recursivily

Extract types by interpreter

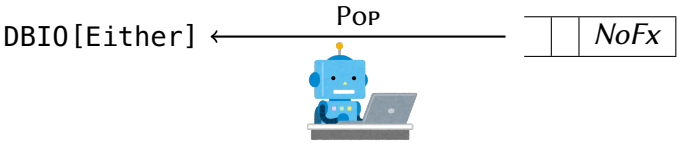
Extract types by interpreter

- *Some* types in the effects stack R can be extracted by *one* interpreter



Extract types by interpreter

- *Some* types in the effects stack R can be extracted by *one* interpreter



- On the other hand, it's good 🦁 that an interpreter doesn't extract just any types from the effects stack

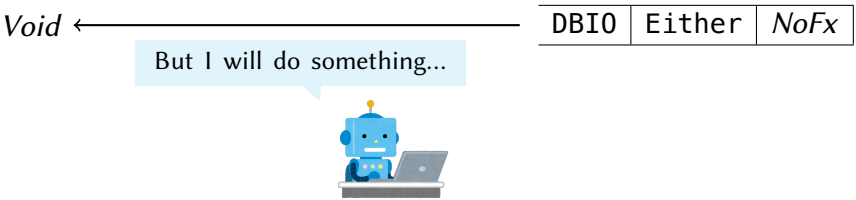


Table of contents

Revisit the problem

- We want to take the both cohesion and consistency between the database transaction and sending e-mails

Revisit the problem

- We want to take the both cohesion and consistency between the database transaction and sending e-mails
- “Over effects” means that
  - There are two effects: the database I/O and sending e-mails
  - If database I/O with trasaction would fail, sending e-mails must *not* be done
  - What an effect should be run depends on that the other effect would be done successfully or not

Create a type constructor

- First we make a type constructor for e-mail

```
sealed trait MailAction[A]
case class Tell(
  mail: Mail
) extends MailAction[Unit]
```

This is just like `Writer` monad, isn't it?

- And we define `DBIOAction` too

```
sealed trait DBIOAction[A]
case class Ask() extends DBIOAction[Session]
case class Execute[A](
  value: A
) extends DBIOAction[A]
```

It's like `Reader` moand, the same as `DBIO`

Utilty functions

- Then we create utilty functions:
  1. First one `sendMailEff` is for making `Eff[R: _mail, Unit]`

```
def sendMailEff[R: _mail](
  mail: Mail
): Eff[R, Unit] = Eff.send[MailAction, R, Unit](Tell(mail))
```

  2. Second one `fromDBIO` is used to convert `DBIO[A]` into `Eff[R: _dbio, A]`

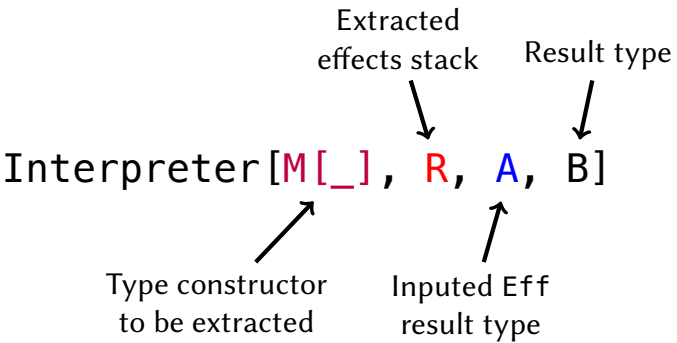
```
def fromDBIO[R: _dbio, A](
  dbio: DBIO[A]
): Eff[R, A] =
  for {
    session <- Eff.send[DBIOAction, R, Session](Ask())
    a <- Eff.send[DBIOAction, R, A](Execute(dbio.run(session)))
  } yield a
```

Type parameters in interpreter

- We have already seen, `Interpreter` (page 22) has such a type parametrs:

```
trait Interpreter[M[_], R, A, B]
```

which mean that 👇

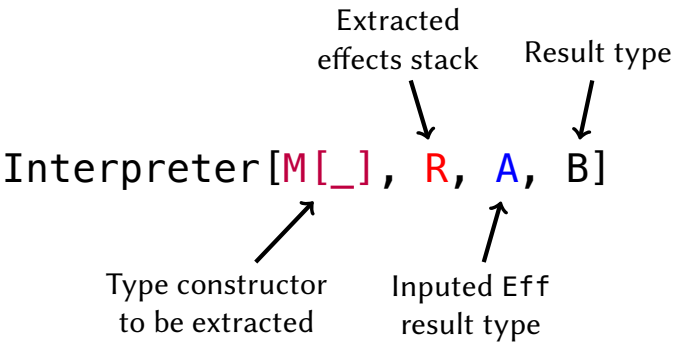


Type parameters in interpreter

- We have already seen, `Interpreter` (page 22) has such a type parametrs:

```
trait Interpreter[M[_], R, A, B]
```

which mean that 👇



It's very complicated 😇
I want to see examples!

Example

```
Eff[R: _dbio: _mail, A]
```

↓

```
Interpreter[DBIOAction, U, A, Either[Throwable, A]]
```

↓

```
Eff[R: _mail, Either[Throwable, A]]
```

- Note that `R: _mail` means `MailAction` is contained in the effects stack `R`
- And `U` is $\boxed{\texttt{Mail} \mid \textit{NoFx}}$. We can calculate `U` from `R` and `DBIOAction` by `Member.Aux[DBIOAction, R, U]`

Implement the interpreter

Implement the interpreter

- It's time to implement the interpreter over effects!

Implement the interpreter

- It's time to implement the interpreter over effects!
- We'll make `DBIOAction` interpreter at first

Implement the interpreter

- It's time to implement the interpreter over effects!
- We'll make `DBIOAction` interpreter at first
- It means that we implement `Interpreter[DBIOAction, U, A, Either[Throwable, A]]` for `Eff[R: _dbio, A]`
  - `U` is the rest of `DBIOAction` extracted from `R`

Implement the interpreter

- It's time to implement the interpreter over effects!
- We'll make `DBIOAction` interpreter at first
- It means that we implement `Interpreter[DBIOAction, U, A, Either[Throwable, A]]` for `Eff[R: _dbio, A]`
  - `U` is the rest of `DBIOAction` extracted from `R`
- Finally `runDBIO` has this interface:

```
def runDBIO[R: _dbio, A](
  eff: Eff[R, A]
)(implicit m: Member.Aux[DBIOAction, R, NoFx]): Either[Throwable, A] =
  withTransaction { session =>
    Eff.run(
      Interpret.runInterpreter(eff)(new Interpreter[DBIOAction, NoFx, A, Either[Throwable, A]] {
        /* We implement now! */
      })
    )
  }
```

1st: onPure

- Following `Interpreter` interface in page 22, we make `onPure` at first

```
def onPure(a: A): Eff[NoFx, Either[Throwable, A]] =
  Eff.pure(Right(a))
```

1st: onPure

- Following `Interpreter` interface in page 22, we make `onPure` at first

```
def onPure(a: A): Eff[NoFx, Either[Throwable, A]] =
  Eff.pure(Right(a))
```

- It's very easy 🙃

2nd: `onEffect`

- Next we implement `onEffect`

2nd: onEffect

- Next we implement onEffect

```
def onEffect[X](
  x: DBIOAction[X], continuation: Continuation[NoFx, X, Either[Throwable, A]]
): Eff[NoFx, Either[Throwable, A]] =
  x match {
    case Ask() => continuation(session)
    case Execute(v) => continuation(v)
  }
```

  - `Continuation[U, X, Either[Throwable, A]]` reperesents a function whose interface is `X => Eff[U, Either[Throwable, A]]`

2nd: onEffect

- Next we implement onEffect

```
def onEffect[X](
  x: DBIOAction[X], continuation: Continuation[NoFx, X, Either[Throwable, A]]
): Eff[NoFx, Either[Throwable, A]] =
  x match {
    case Ask() => continuation(session)
    case Execute(v) => continuation(v)
  }
```

  - `Continuation[U, X, Either[Throwable, A]]` reperesents a function whose interface is `X => Eff[U, Either[Throwable, A]]`

- First we have to use the pattern matching for `x` to determine what `X` is

  `Ask case` `X` is `Session`, and `continuation` is needed that value

  `Execute case` we don't know what `X` is, but `Execute` has `X` value

2nd: onEffect

- Next we implement onEffect

```
def onEffect[X](
  x: DBIOAction[X], continuation: Continuation[NoFx, X, Either[Throwable, A]]
): Eff[NoFx, Either[Throwable, A]] =
  x match {
    case Ask() => continuation(session)
    case Execute(v) => continuation(v)
  }
```

  - `Continuation[U, X, Either[Throwable, A]]` reperesents a function whose interface is `X => Eff[U, Either[Throwable, A]]`

- First we have to use the pattern matching for `x` to determine what `X` is

  `Ask case` `X` is `Session`, and `continuation` is needed that value

  `Execute case` we don't know what `X` is, but `Execute` has `X` value

- We can access `session` because the interpreter is in `withTransaction` (page 33)

3rd: onLastEffect and onApplicativeEffect

- These are just type puzzle

```
def onLastEffect[X](x: DBIOAction[X], continuation: Continuation[NoFx, X, Unit]): Eff[NoFx, Unit] =
  x match {
    case Ask() => continuation(session)
    case Execute(v) => continuation(v)
  }

def onApplicativeEffect[X, T[_]: Traverse](
  xs: T[DBIOAction[X]], continuation: Continuation[NoFx, T[X], Either[Throwable, A]]
): Eff[NoFx, Either[Throwable, A]] =
  continuation.apply(
    xs.map {
      case Ask() => session
      case Execute(v) => v
    }
  )
```

3rd: onLastEffect and onApplicativeEffect

- These are just type puzzle

```
def onLastEffect[X](x: DBIOAction[X], continuation: Continuation[NoFx, X, Unit]): Eff[NoFx, Unit] =
  x match {
    case Ask() => continuation(session)
    case Execute(v) => continuation(v)
  }

def onApplicativeEffect[X, T[_]: Traverse](
  xs: T[DBIOAction[X]], continuation: Continuation[NoFx, T[X], Either[Throwable, A]]
): Eff[NoFx, Either[Throwable, A]] =
  continuation.apply(
    xs.map {
      case Ask() => session
      case Execute(v) => v
    }
  )
```

- I know that it's very diffcult for us but applicative interpreter is not the scope in this talk so we'll skip

Interpreter for `MailAction`

Interpreter for `MailAction`

- We have already gotten the interpreter for `DBIOAction` so then we'll create a new interpreter for `MailAction`

Interpreter for `MailAction`

- We have already gotten the interpreter for `DBIOAction` so then we'll create a new interpreter for `MailAction`
- The interface is `Interpreter[MailAction, U, A, (List[Mail], A)]` for `Eff[R: _mail, A]`

What is `(List[Mail], A)`?

Interpreter for `MailAction`

- We have already gotten the interpreter for `DBIOAction` so then we'll create a new interpreter for `MailAction`
- The interface is `Interpreter[MailAction, U, A, (List[Mail], A)]` for `Eff[R: _mail, A]`

What is `(List[Mail], A)`?

- We try to
  1. collect e-mails by a new interpreter we'll create from now
  2. execute `runDBIO` then if the result is successful run `sendMail` with 1's e-mails. However if the result is failure `sendMail` is not call and 1's e-mails will not be sent

Interpreter for `MailAction`

- So `sendMailAfterDBIO`'s interface:

```
def runMailAfterDBIO[R: _mail: _dbio, U, A](
  eff: Eff[R, A]
)(
  implicit m1: Member.Aux[MailAction, R, U], m2: Member.Aux[DBIOAction, U, NoFx]
): Either[Throwable, A] = {
  val mailRemoved: Eff[U, (List[Mail], A)] =
    Interpret.runInterpreter(eff)(new Interpreter[MailAction, U, A, (List[Mail], A)] {
      /* We implement now! */
    })

  runDBIO(mailRemoved).flatMap {
    case (mails, a) =>
      mails.traverse(sendMail).map(_ => a)
  }
}
```

  - Note that `sendMail` is defined on 12 page

Interpreter for `MailAction`

- This is it!

```
trait Interpreter[MailAction, U, A, (List[Mail], A)] {
  def onPure(a: A): Eff[U, (List[Mail], A)] =
    Eff.pure((Nil, a))

  def onEffect[X](
    x: MailAction[X], continuation: Continuation[U, X, (List[Mail], A)]
  ): Eff[U, (List[Mail], A)] =
    x match {
      case Tell(mail) =>
        // `Tell extends MailAction[Unit]` so in this case `X` is `Unit`
        continuation(()).map {
          case (mails, a) => (mail :: mails, a)
        }
    }
}
```

  - `onLastEffect` and `onApplicativeEffect` are omited from the slide 😇

Usage

- We can use this like below:

```
def userUpdateEff[R: _dbio: _mail](
  newUserInfo: UserInfo
): Eff[R, Unit] =
  for {
    _ <- fromDBIO(userUpdate(newUserInfo))
    _ <- sendMailEff(Mail(/* very great email from `newUserInfo`*/))
  } yield ()
```

```
val user: UserInfo = ???

runMailAfterDBIO(userUpdateEff(user)) match {
  case Right(_) => /* Succecss DB and e-mail! */
  case Left(_)  => /* Failure */
}
```

Usage

- We can use this like below:

```
def userUpdateEff[R: _dbio: _mail](
  newUserInfo: UserInfo
): Eff[R, Unit] =
  for {
    _ <- fromDBIO(userUpdate(newUserInfo))
    _ <- sendMailEff(Mail(/* very great email from `newUserInfo`*/))
  } yield ()
```

```
val user: UserInfo = ???

runMailAfterDBIO(userUpdateEff(user)) match {
  case Right(_) => /* Succecss DB and e-mail! */
  case Left(_)  => /* Failure */
}
```

- It looks good, doesn't it?🤔

Discussion: Monad vs Eff

[†]but I cannot explain this due to the time limit of this talk 😇

Discussion: Monad vs Eff

This example can be done by monad (transformer)?

[†]but I cannot explain this due to the time limit of this talk 😇

Discussion: Monad vs Eff

This example can be done by monad (transformer)?

- That's correct, but
  - in this example only covers a case in which, "if the transaction fails no e-mail is sent"
  - other cases could exist; for example, maybe we would like to send error e-mails when the transaction fails

[†] but I cannot explain this due to the time limit of this talk 😇

Discussion: Monad vs Eff

This example can be done by monad (transformer)?

- That's correct, but
  - in this example only covers a case in which, "if the transaction fails no e-mail is sent"
  - other cases could exist; for example, maybe we would like to send error e-mails when the transaction fails
- If we do that with monads, we cannot make it without changing interfaces to distinguish whether an e-mail will be sent or not when a transaction fails

[†] but I cannot explain this due to the time limit of this talk 😇

Discussion: Monad vs Eff

This example can be done by monad (transformer)?

- That's correct, but
  - in this example only covers a case in which, "if the transaction fails no e-mail is sent"
  - other cases could exist; for example, maybe we would like to send error e-mails when the transaction fails
- If we do that with monads, we cannot make it without changing interfaces to distinguish whether an e-mail will be sent or not when a transaction fails
- Eff can do that without changing any interfaces, we only change the interperter.†

†but I cannot explain this due to the time limit of this talk 😇

Discussion: Monad vs Eff

- Or, we can call `sendMail` outside of monads

```
val dbioMail: Writer[List[Mail], DBIO[?]] = ???

val (mails, dbio) = dbioMail.run // `Writer` run
withTransaction(dbio.run) match {
  case Right(_) => List.traverse(mails)(sendMail)
  case Left(_)  => // error!
}
```

Discussion: Monad vs Eff

- Or, we can call `sendMail` outside of monads

```
val dbioMail: Writer[List[Mail], DBIO[?]] = ???

val (mails, dbio) = dbioMail.run // `Writer` run
withTransaction(dbio.run) match {
  case Right(_) => List.traverse(mails)(sendMail)
  case Left(_)  => // error!
}
```

- Indeed it can be done but it's outside of monad, so it's maybe not succesful to reprensenting effects by monad 🤔

Conclusion

Conclusion

- In this talk, we see that some ways to database I/O and sending e-mails

Conclusion

- In this talk, we see that some ways to database I/O and sending e-mails
- Monad types are embedded its concrete operation for the effect but Eff is not. Types are just symbols and the concrete operation is given by the interpreter

Conclusion

- In this talk, we see that some ways to database I/O and sending e-mails
- Monad types are embedded its concrete operation for the effect but Eff is not. Types are just symbols and the concrete operation is given by the interpreter
- Therefore an interpreter can do the complex operation which is over some effects

Conclusion

- In this talk, we see that some ways to database I/O and sending e-mails
- Monad types are embedded its concrete operation for the effect but Eff is not. Types are just symbols and the concrete operation is given by the interpreter
- Therefore an interpreter can do the complex operation which is over some effects
- Let's use Eff!

References

[1] Oleg Kiselyov and Hiromi Ishii.
Freer monads, more extensible effects.
https://www.slideshare.net/konn/freer-monads-more-extensible-effects-59411772, 2016.

Thank you for your attention!