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）

エフェクトを跨ぐインタープリター

- こっちのページは日本語の説明となります。
- 翻訳の方向けの情報もはいっているので、多少は冗長かもしれないです。

Table of contents

- このトークでは、まずモナドよりも前の低レイヤーな方法を紹介しつつ、それの問題とこれまでどういう方法で解決してきたか？ ということを通してモナドなどについて解説する。そのあと Eff とインタープリターの説明をして最後にこのトークのテーマである「エフェクトを跨ぐインタープリター」を実装する。

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

- リクルートマーケティングパートナーズに中途入社した。
- スタディサプリ ENLGISH のサーバーサイドを Scala で作っている。
- 量子コンピューター（*Quantum Information*）[a]についても興味がある。
- 暗号やセキュリティーについても好きで、今も少しは CTF[b]に出場したりもする。
- 最後にこのスライドを作るのにも使った LaTeX も非常に好きで、実は Scala 以上に利用歴がある。

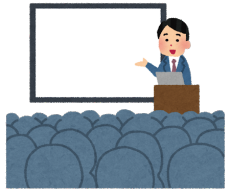
---

[a] ふつう量子コンピューターは “Quantum Computer” となるが、Quantum Computer であると物理的なハードウェアなニュアンスもあるため、そうではなくて量子コンピューター上で動作する（予定の）ソフトウェアまたはプロトコルを意図して量子情報（“Quantum Information”）とした。（必要ないかも🤔）

[b] “Capture The Flag”、セキュリティー系の競技のこと。

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

- このトークでは、全体を通して 1 つの具体的なケースである、“データベースにトランザクションを張りながら読み書きをする” という話を例にする。
- プログラミングにおいて、これはかなりありふれたケースだが、いろいろなやり方がある。
- 低レベルなやり方やそれの問題を述べつつ、最終的にそういった問題が `Eff`でどう解決されるのか？ について説明していく。

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`?

- まず念のために言うとしたら、低レベルというのが常に悪いということではない。
- ただ考えないといけないことが増えたり、考え忘れたりするとひどいことになりやすい。考え忘れるというのがどういうことなのか？ それを具体的にみていく。
- 今このようなコードでトランザクションをかけながら DB へのアクセスをする。
- `TM`は“トランザクションマネージャー”のことで、これが DB のトランザクションを管理して、`begin`・`commit`でトランザクションの開始・終了を表す。
- 冒頭に言った「考えわすれる」というのは、このコードで言えば`begin`・`commit`を忘れるということ。

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?

- この「忘れる」問題への対処として“ローンパターン”が知られている。
- このように関数を取る関数 `withTransaction`に関数 `f`を渡してそれをトランザクション内で実行する
- こうすればデータベースのトランザクションのように「借りたもの（Loan）」を返し忘れることがない[a]。
- こうすれば少なくとも忘れる問題には対処できるが、はたしてこれだけで OK だろうか？

---

[a] トランザクションを著者が完全に理解しているわけでもないが、データベースの一貫性などを担保する目的で一時的に利用するものであって、永遠に利用することはできないので、いずれは返す（解放する）必要があるということを言いたい。

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

- ただ Loan pattern はこのように無理やり入れ子にして使うこともできてしまう……。
- こうやって使いたい人は恐らく誰もいないが、しかし型システム（*type system*）上はこのようなコードが許可されてしまう。
- たしかに忘れるということは防げるものの、問題は他にも残っている。
- ただ、この入れ子になってしまう問題は一番最初に紹介した低レベルの方法でも起きることであって、Loan pattern はそれよりは少なくとも安全にはなっているはずである。

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! */)
  }
```


- そこで Loan pattern を生で使うのではなくて、`map`と `flatMap`でラップ（wrap）する。
- このように `DBIO`[a]というデータ構造をつくって、それに `map`・`flatMap`を実装する。
- さらにユーティリティーとして `ask`というのをこのように作っておく。
- そうすれば DB にアクセスするようなコードを `ask`でこうやって書くことができる。

[a] *Database Input/Output* の略。

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]`

- あとは 1 つのトランザクションに入れたいコードを `for`で合成（composition）していけばいい。
- 最後に `DBIO`の `run`を Loan pattern 用の関数 `withTransaction`の中で実行してやればいい。
- もちろん `DBIO`の中で `withTransaction`することもできるが、型として “DB にアクセスする操作の型は `DBIO`である” となっているので、これを入れ子にしてしまうことは多少防がれるはずである。
- ところで `DBIO`は新規で作ったが、同じことができるよく知られたモナドがあるのではないだろうか？
- それは正解で `DBIO[A]`は環境（*environment*）を `Session`に固定した Reader モナドである `Reader[Session, A]`と同じような感じである。

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*

- `DBIO`はデータベース I/O というただ 1 つを表現するが、他の副作用を利用したいときもある。
- そこで次は、データベーストランザクションが成功したときにメールを送信するというシチュエーションを考えていく。

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

- メールを送るというのもまた、データベースを使ったプログラミングと同じくらいよくある処理である。
- 必要ないかもしれないが、
- データベースとの大きな違いとして、メールの送信にトランザクションは**存在しない**。あたりまえのことだが、メールを送った後に後続の処理が失敗したとしても、いったん送信してしまったメールを取り消すことはできない。
- したがってメールの送信は全ての処理が正常に終了となったあとに行いたい。

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

- まずメールを送る関数 `sendMain`を次のように定義する。成功したら `Either.Right` で失敗したら `Left`となる。
- さて、一番ナイーブにはこのように `withTransaction` の後で DB セッションの状態をチェックしつつ実行すればよい。
- 動きそうだが、これはちょっと `sendMain`とデータベース処理の間のコード上の距離が離れすぎてないか……？

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

- コードの距離とはどういうことか？ ようするに凝集度（*cohesion*）のこと。
- `DBIO`を使った関数を書くが、この場所にメールの送信処理を書くことはできない。
- しかしユーザーの変更処理はメールを送信することも 1 つのビジネスロジックであるから、DDD 的にどこでやるか？ などはいったん放置するにしてもビジネスロジックの近辺にあった方が凝集度が高くなりそうだ。
- でも我々の今の方法では`withTransaction`のところに書かざるを得ない……

Cohesion

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

```scala
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

- そういう問題があるので、じゃあ`DBIO[Either[Throwable, A]]`を使ったらどうか？
- これはコード例のように`DBIO.ask.map`の中で`Either`を返すようにする。このようにすれば、`DBIO`を利用するところでメールの送信ロジックを記述できる。
- したがって凝集度は向上すると考えられるが、これでいいだろうか？

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`

- いったんこのコードが意図した動作をするか？ というところは放置して、このようにモナドトランスフォーマー（*Monad transformer*）を利用することもできる。
- モナドトランスフォーマーは、モナドとなる型を取って（*take*）モナドとなるような高階（*higher kind*）な型コンストラクターである。
- ここでは `EitherT`の詳細を説明しないが、このようにモナド `DBIO`を引数にとって `DBIO`と `Either`の両方の能力を持つ新たなモナドとなる。
- Scala の `for`式（*expression*）一番外側のモナドにしかアクセスできないため、モナドトランスフォーマーを利用して 1 つにすると取り扱いしやすくなる。

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!

- もうすでに気がついているかもしれないが、このように `for`で合成したときに後続の処理の `DBIO`で失敗したとしても、メールは送信されてしまう。
- `DBIO`が失敗したということはデータベースはロールバックで操作がすべてなかったこととなる。しかしメールは途中までの処理が成功した！ という内容で送信される。
- したがってこれはメールとデータベースの間で**不整合**（*inconsistency*）が生じた状態であり、避けるべきである。

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

- ここまでのまとめをすると、凝集度と一貫性（*consistency*）を両立したいが、しかしここまでに紹介した技術では「どちらを取るか？」というトレードオフになる。
- 自分の知る限りで、この問題への対処方として 2 つの方法があり、
  1. 専用のオリジナルモナドを作ってしまうこと
  2. `Eff`と専用のインタープリターをつくること

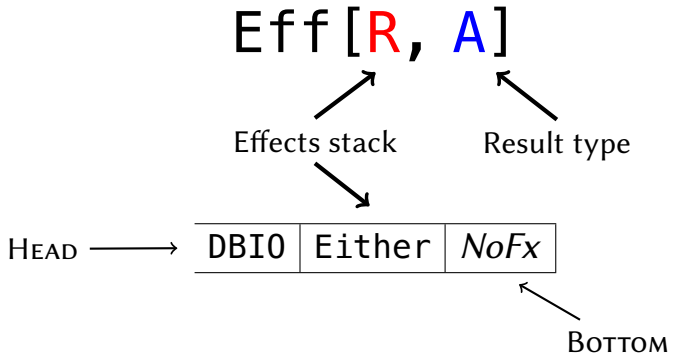
  それがこの☝ 2 つである。
- まずここから2の話しをしていきそのあとで自分の考えとして1と2でどちらがよいと思っているのか？ を話すことにする。

Table of contents

- ここまでで目次にあった `Eff`以外の部分の説明が終わったので、ここからようやく `Eff`の説明にはいっていく。

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`.

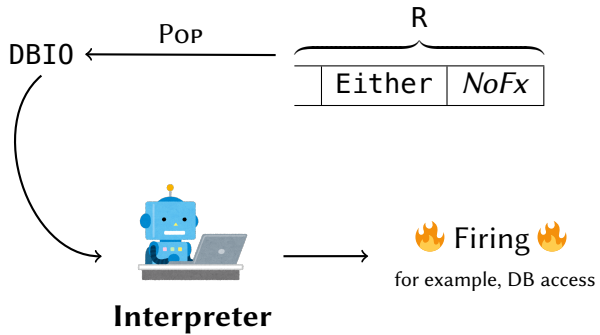
- `Eff`は 2 つの型パラメーター `R`と `A`を取るような型コンストラクターで、`R`が**エフェクトスタック**（*effects stack*）[a]を表し、`A`が結果の型（*result type*）を表す。
- 簡単のために、このトークではエフェクトスタックはこのような型レベルスタック[b]であるということにしておく。
- いま、図では先頭が `DBIO`で次に `Either`があり最後に *NoFx* となっている。この *NoFx* はリストでいうところの `Nil`にあたるものである。

[a]エフェクトは一般的に「副作用」と考えてもらえばよいと思う。副作用というと語感から「よくない」というニュアンスを与えるが、実際上は副作用のないプログラムはたいしたことができない。したがって副作用をうまく隠蔽（？）しつつ使うのが重要であり、その観点からより公平というか中立な語として “computational effect” やさらに省略して “effect” というふうに呼ばれるようなったと、著者は認識している。

[b]筆者の慣習上、型レベルスタックを “型レベルリスト” と言うことがあるが、この 2 つは両方おなじようなものなので翻訳ではどちらも “type level stack” で問題はない。

Interpreter

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

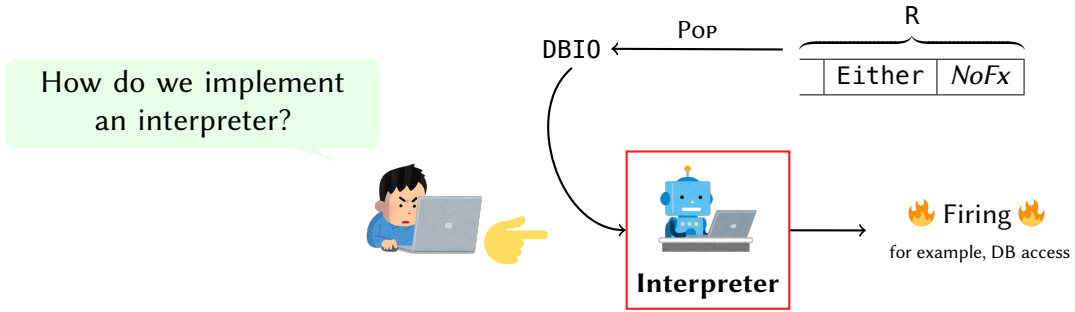


- It means that types in R are just “symbols” so they don’t have the logic for real effects
  - Firing effect logics are given by interpreters

- そして、このようにエフェクトスタックにある各型を処理するためのインタープリター（*interpreter*）が必要になる。
- このインタープリターがエフェクトスタック `R`から `DBIO`など型を取り出して、そして実際のエフェクトを発生させる。
- したがって `R`に入っている型はあくまでも「記号」でしかなく、この型に具体的なエフェクトを発生させるロジックは埋めこまれていない。エフェクトを実際に発生させるのはあくまでもインタープリターである。

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



- ここまで説明してきた関係は依存性の注入（*dependecy injection*）と似ている。DI ではインターフェースを定義してそれに具体的な実装を与える。インターフェースには具体的な処理を書いていないので、たとえばデータベースにアクセスするインターフェースであっても MySQL 用の実装と Postgress 用の実装を 2 つ用意して場合によって使いわけたりできる。
- これと同様に Eff ではエフェクトスタック上の型に対応するインタープリターがあり、それが具体的に計算作用（computational effect）を発生させる。DI と同様に、エフェクトスタックに入っている個々の型には、実際に発生する作用の具体的な実装は記述されておらず、それらはインタープリターで行なう。
- さて、ここまでで Eff とそのインタープリターについて説明したところで、このインタープリターをどうやって書いていくのか？ というところを説明していきたい。

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]
}
```

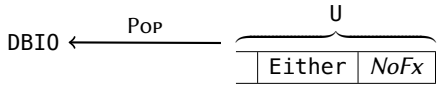
  - https://github.com/atnos-org/eff/blob/master/shared/src/main/scala/org/atnos/eff/Interpret.scala

What does it mean?

- とりあえず atnos-eff の `Interpreter`インターフェースをもってきた
- これを見ても一瞬ではさっぱりわからないと思うので、ここから直感的なことをもう少しくわしく説明していく。

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?

- いま、`DBIO`のインタープリターを `Eff[R, A]`、ただし `R`は | `DBIO` | `Either` | *NoFx* | という状況について考えていく。
- 図のように `R`から `DBIO`が取り出される。ここで残ったエフェクトスタックを `U`と呼び | `Either` | *NoFx* | である。
- このとき、インタープリターはモナドインスタンスの実装のために次の 2 つを提供してくれる。この例でいえば、これらを使って我々はモナドインスタンスのための `map` と `flatMap`をあたえなければならない。
- ところで、ここでいう❷の継続（*continuation*）とはいったいどういうことだろうか？

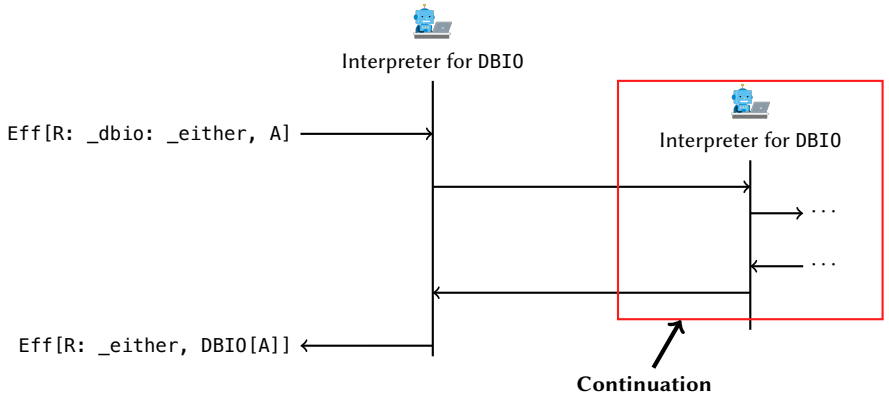
Notation

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

- 説明のまえに、エフェクトスタックをのための新しい表記を導入する。
- いま `R: _dbio: _either`と書いたら、これは Rが | `DBIO` | `Either` | *NoFx* | というエフェクトスタックであることを意味する。このようにすると `Eff[R: _either, A]`などと書けて便利である。

Continuation for interpreter

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

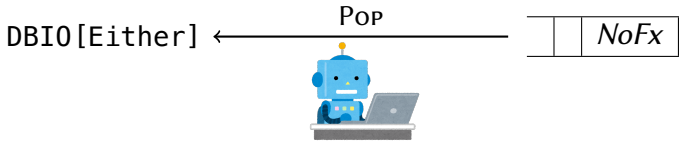


- First interpreter can access the continuation as a function, which processes effect recursivily

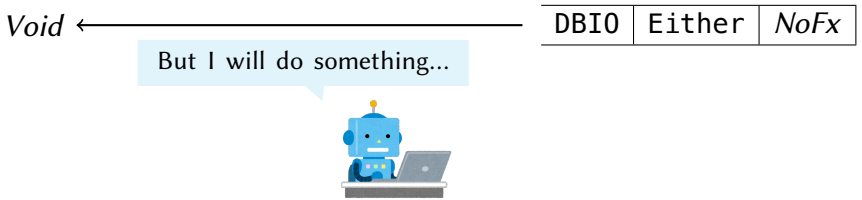
- このように合成されたインタープリターが処理するときには、現在処理しているモナドよりも後の部分を継続として得ることができる。
- そして全ての処理が終了するとエフェクトが実際に発生して、エフェクトスタックにあった型が結果側へ移動する。

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



- また、これまでの例では 1 つのインタープリターが 1 つの型をエフェクトスタックから取り出して処理をするという説明をしてきたが、実はこのように 2 つ以上の型を 1 つのインタープリターで同時に処理してもよい。
- さらに 0 個処理する、つまりはエフェクトスタックから何も取り除かないようなインタープリターも OK である。

Table of contents

- さて、ここまでで Eff とそのインタープリターの紹介が終ったので、ここからはさきほどまで見てきた具体的な問題である「DB アクセスとメール送信」の話題に戻っていく。

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

- 我々はメール送信とデータベースのトランザクションにおける凝集度と一貫性の両方を達成したい。
- タイトルにある “over effects” とはこのように 2 つのエフェクトが相互に関係して「片方が成功したらもう片方をやる」といったことを記述することを意図している。

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`

- まずメールを `Eff`のエフェクトスタックに載せるためにこのような型コンストラクター `MailAction`を作っておく。
- sealed trait になっていて、唯一の値として `Tell`を持つ。これは中身に `Mail`な値を持つ。
- これが `Writer`モナドに見える（？）かもしれないが、それの理由は後に解説する。
- 次に `DBIOAction`という DB 用の型を定義する。`DBIO`がほぼ `Reader`モナドであったように、Eff でも似たような構造になる。

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
```

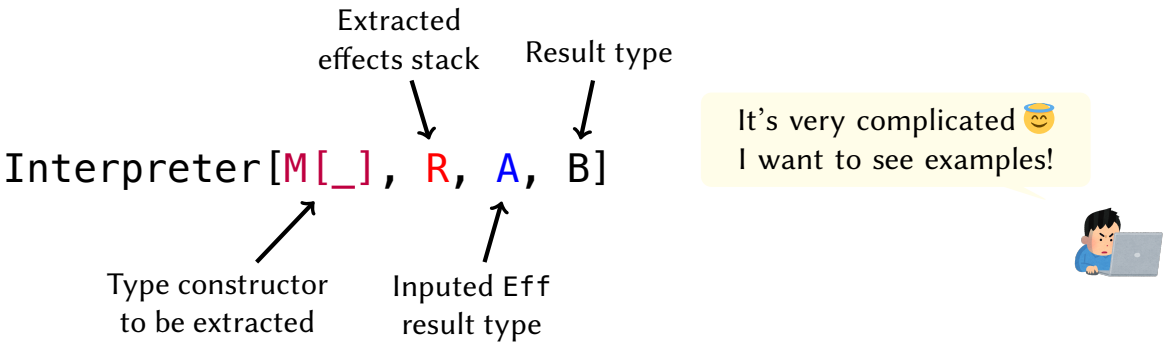

- まずは `Eff[R: _mail, Unit]`を作成するための `sendMailEff`と既存の `DBIO[A]`を `Eff[R: _dbio, A]` へと変換する関数 `fromDBIO`を作成する。
- これらは atnos-eff で定義されている `Eff.send`を使って実装する。
  - この `Eff.send`をきちんとコードで追うためには、さきほど説明した `Eff`のエフェクトスタックが実際にどう表現されているのか？ ということを詳細に理解する必要があるため、いったんここではあえて説明しない。
  - とりあえずのところは、型コンストラクターに包まれたような `M[A]`において、この `M`をエフェクトスタックへ適切に追加してくれるものという理解でよいと思う。
- `DBIO[A]`から `Eff[R: _dbio, A]`を作るのは少し大変となる。
- まずは `Ask`を使って `Eff[R: _dbio, Session]`をとする。そして `for`の中でまわすことで `session: Session`を得る。これで `dbio.run(session)`を `Execute`の中で行なえば狙った型にすることができる。

Type parameters in interpreter

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

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

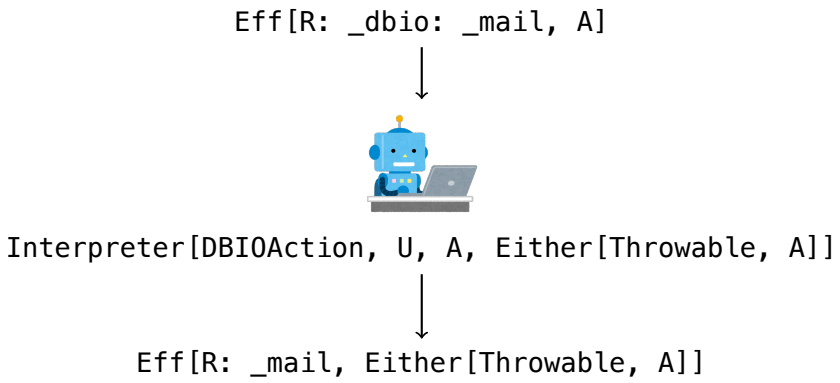
which mean that 👇



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

- ここでは先程みた `Interperter`の各型パラメーターについて説明していく。
- この図の通りになっているが、複雑というかイメージが掴みにくいかもしれないのでさっそく例を見ていく。

Example



- Note that `R: _mail` means `MailAction` is contained in the effects stack `R`
- And `U` is `Mail` | *NoFx*. We can calculate `U` from `R` and `DBIOAction` by `Member.Aux[DBIOAction, R, U]`

- このようにまず `R`が `DBIOAction` | `Mail` | *NoFx* という `Eff[R: _dbio: _mail, A]` が、このように型パラメーターを入れたインタープリターに入力するとする。
- すると図の下にあるような型の `Eff`となる。
- また、エフェクトスタック `R`から `DBIOAction`を取り除いた残りのエフェクトスタックは `Member.Aux[DBIOAction, R, U]`という型クラス（*type class*）によって型 `U`で得られる。

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! */
      })
    )
  }
```

- ここからいよいよインタープリターを実装していく。まず`DBIOAction`のインタープリターを実装する。
- 下のコードを見てほしいが、このインタープリターは`Member.Aux[DBIOAction, R, NoFx]`のインスタンスを取ることによってエフェクトスタックが`DBIOAction`しかない状態を強制している。エフェクトスタックが空（*NoFx*）のとき、`Eff.run`によって`Eff[R, A]`の`A`を取得できる。
- そして、このようにインタープリターの定義を全部ローンパターンのときに作った`withTransaction`の中に押し込んでいる。これは後で重要になるが、これによってインタープリター内部で`session`にアクセスできるようになる。

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 🙃

- このページのインターフェースにしたがって、まずは `onPure`をつくっていく。といってもこれはモナドの `pure`が簡単なようにすごく簡単に作ることができる。

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)

- 次は onEffectである。このケースでは x: DBIOAction[X]に対してパターンマッチをする。

- Ask のとき 外にある withTransactionで捕獲しておいた sessionがあるので、それを continuationにいれることで継続を実行できる
  Execute のとき Executeが持つ Xの値を使って継続を実行できる

- そして Continuation[U, X, Either[Throwable, A]]は
  X => Eff[U, Either[Throwable, A]]みたいな関数のインターフェースを継承しているので、ここに上で得た X型の値を入れてることで onEffectの返り値と同じ型を得ることができる。

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

- ここは atons-eff の `Interpreter`のインターフェース上必要なので作っただけという側面が強い。おそらくアプリカティブのときには `Future`を並列にするとかそういうパフォーマンスの向上のためにある気がする。
- そういう理由なので、いったんこれの説明は割愛する。

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

- そういうわけで `DBIOAction`のインタープリターが完成したのでつぎにメールのやつをつくる。
- 定義時にも述べたように、これのインタープリターはこのように結果の型が `Writer`モナドのタプルのようになっている。
- これはどういうことかというといったんはこのように送信するメールを `List[Mail]`として蓄えておいて、そしてそのあとで `runDBIO`を実行してそれが成功していた場合はそのリストの内容を全て送信処理に回すという作戦である。
- もしデータベース処理で失敗がおきていれば、メールのリストは破棄してメールを送信しない。

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

- そういうわけで DB 処理を実行して成功していたらメールを送る関数 `runMailAfterDBIO`はこのようになる。
- 内部で `runDBIO`を使う関係で、これもエフェクトスタックがちょうど

`MailAction`	`DBIOAction`	*NoFx*

であるように `Member.Aux`で調整している。

- まずはこれから作るメール用のインタープリターでメールと結果を分離してタプルにする。そのあとで `runDBIO`の結果をみて実際にメールを送信する `sendMail` を呼ぶかどうか？ を決定する。

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 😇

- そしてインタープリターがこれ！
- `onEffect`のところで、いま`MailAction`は型パラメーターとして任意の型を取ることができる。しかし唯一の値である`Tell`は`MailAction[Unit]`で固定されているため、したがってここに来た時点で`X`は`Unit`ということになる。あとは継続を呼び出しつつメールをリストに追加するだけである。
  - ここでは`Tell`しかパターンがないものの、`X`が`Unit`であることをコンパイラーに教えるためにパターンマッチをしておく。
- さきほどの理由があって`onLastEffect`と`onApplicativeEffect`は省略する。

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?🤔

- さて、実際完成したインタープリターなどはこのように使うことができる。`userUpdateEff`は見てのとおり、ユーザーのアップデート処理のすぐあとにメール送信処理が書いてあり、見通しがよいと思われる。

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 😇

- ここまでの例を見て、これはモナドやモナドトランスフォーマーでも達成できたのでは？ というふうに思うかもしれない。
- 実はこの例だけだとそういうことになるが、たとえばさらに複雑にする方法として
  - 今回はトランザクションが失敗したらメールは全て破棄という方針であったが、実際にはエラーメッセージの送信など一部のメールは送るという場合もある。
  - そういうときにモナドトランスフォーマーのアプローチは型（インターフェース）を変更して「失敗したときに破棄するところ」と「失敗したら送るところ」を分ける必要があると思う。
- 一方で Eff はそのような変更なしで特定の部分の変更をなかったことにする（継続を適用しない）といったことがインターフェースの変更なしで達成できる。
  - ちょっと時間がないのでここでは説明しないけど……。

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 🤔

- このようにモナドの外側に処理を書くことで、失敗したときにもメールを送るか送らないか、を制御できることはできる。
- たしかにこれで達成できてはいるが、これはあくまでモナドの外側で行なわれたことであり、つまり作用（エフェクト）をモナドで表現するということに完全に成功しているのか？ というと個人的には疑問が残る。

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!

- というわけで、このトークではデータベース I/O のいろいろなやり方を紹介した。
- モナドはその型自体に具体的なエフェクトのためのオペレーションが記述されているが、一方で Eff の場合、型は単なるシンボルであって実際の操作はインタープリターが持っている。
- それによって複数のエフェクトを跨ぐような複雑な操作を行うことができる。
- Eff を使っていこう！

References

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

- この資料は Oleg さんと論文と同名だが、Eff 論文の共著者である石井さん日本語による解説発表となっている。
- ScalaMatsuri には日本人の方が多いと思うので、ソースコードは Haskell となってはいるが、この資料がとても参考になると思う。
- このスライドを執筆するときにもおおいに参考になった。

Thank you for your attention!