
FUJITASK MEETS EXTENSIBLE EFFECTS

Hikaru YOSHIMURA (吉村 優)

Recruit Marketing Partners Co., Ltd.

yyu@mental.poker

ScalaMatsuri on June 29, 2019

(y-yu/fujitask-eff-slide@249be0e)

目次

- ① 自己紹介
- ② トランザクションとは？
- ③ トランザクションのやり方
- ④ モナド版 **FUJITASK**
 - **FUJITASK** とサブタイプ
- ⑤ Extensible Effects
- ⑥ まとめ

自己紹介



- 筑波大学 情報科学類卒（学士）
- プログラム論理研究室
- トランザクションとサブタイプについて

Twitter @_yyu_
Qiita yyu
GitHub y-yu

トランザクションとは？

● トランザクションは次を満たす

- 全操作が完了するまで、他のプロセスはその途中の状態を観測できない
- いずれかの操作が失敗した場合は全てが失敗となり、データベースは操作を行う前の状態に戻る

- たとえば「課金したならばコーチング機能を使えるようにし、コーチをアサインする」という次のような処理がある
 - ① 課金履歴の更新
 - ② コーチング機能を使えるという権限管理の更新
 - ③ コーチのアサイン情報の更新
- これらのうちどれかひとつでも失敗したならば中途半端な状態にはならず、なにもしなかった状態となる

トランザクションのやり方

- ① BEGIN と COMMIT に相当する関数を用意してそれらを実行する

```
val transactionManager = new TM()  
  
transactionManager.begin()  
  
something.databaseOperation()  
  
transactionManager.commit()
```

- beginとか commitを忘れたら終わり
- C 言語といった表現力の低いプログラム言語では、このようなスタイルしかできなかった
 - たとえば、かつてのファイル操作はこんなシンタックスでロックするなどしていた
- 最近のプログラム言語でこのような方法を使うことは少ない

トランザクションのやり方

② 高階関数を利用する（ローンパターン）

```
def withTransaction(f: Session => Unit): Unit = {  
  val transactionManager = new TM()  
  
  transactionManager.begin()  
  f(transactionManager.session)  
  transactionManager.commit()  
}  
  
withTransaction { session =>  
  something.databaseOperation(session)  
}
```

- 関数を引数として渡せるプログラム言語ではよく利用される
 - 関数が引数として渡せなくとも、関数を表すようなインターフェースを利用することで昔のJavaなどでも利用できた
- 最近のプログラム言語ではこの方法を使うことが多い（？）

トランザクションのやり方

- ② のローンパターンを使えば解決か？
 - とはいえ、withTransactionを忘れたら終わり
- さらに、次のようにプログラマーがミスで二重にトランザクションを貼ってしまったらどうなるだろう？

```
def somethingOperation(): Unit = {  
  withTransaction { something.databaseOperation }  
}  
  
withTransaction { somethingOperation }
```

- さらに、2つのトランザクション処理を結合したくなくとも、新しく処理を書かなければならない
- そもそも、多くの場合トランザクションが必要かどうかはSQLから判定できる
- にも関わらず、ローンパターンではSQLを使う時にトランザクションが必要かを判断している

FUJITASK とは？

FUJITASK

データベースのトランザクションを管理するデータ構造（モナド）

- トランザクションを貼るかどうか？ を**コンパイル時**に判断してくれる
- プログラマーの明示的な操作なしに適切なトランザクションの開始と解放が実行され、かつトランザクションが二重・三重になったりしない
- かつてドワンゴにいた藤田さんが開発したため、このように呼ばれている

FUJITASK とは？

- 定義は次となる

```
trait Task[-R, +A] { lhs =>
  def execute(r: R)(implicit ec: EC): Future[A]

  def flatMap[ER <: R, B](f: A => Task[ER, B]): Task[ER, B] =
    new Task[ER, B] {
      def execute(r: ER)(implicit ec: EC): Future[B] =
        lhs.execute(r).map(f).flatMap(_.execute(r))
    }

  def map[B](f: A => B): Task[R, B] = flatMap(a => Task(f(a)))

  def run[ER <: R]() (implicit runner: TaskRunner[ER]): Future[A] =
    runner.run(lhs)
}
```

- flatMapに**サブタイプ**を使っている！
- Haskell にはサブタイプが無いとよく知られているが、なぜこのモナドはサブタイプを使っているのか？

サブタイプと半順序集合

サブタイプ

型 B が型 A の期待されている場所で安全に使用可能であるとき、 B は A の**サブタイプ**である

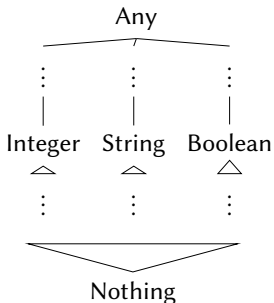
- 型 B が型 A のサブタイプであるとき $B <: A$ と表記する
- サブタイプ関係には次のような特徴がある
 - ① 任意の型 A において、 $A <: A$ である（反射）
 - ② 任意の型 A, B, C において、 $C <: B$ かつ $B <: A$ ならば $C <: A$ である（推移）
 - ③ 任意の型 A, B において、 $B <: A$ かつ $A <: B$ ならば $A = B$ である（反対称）
- このような関係と集合を**半順序集合**と言い、型とサブタイプ関係は半順序集合となる

サブタイプと束

- Scala のサブタイプ関係は**束**を作る

束

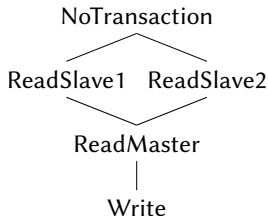
任意の 2 つの元が一意的な上限および下限を持つ半順序集合のこと



- どんな 2 つの元も、一意的な共通の祖先と子孫をそれぞれ持つ
- 型システム理論においては、ボトム (**Nothing**) を入れると複雑になると知られており避けられる傾向にあるが、Scala には入っている
- そのため、Scala のサブタイプ関係は**束**となる

トランザクションと束

- 実はトランザクションは束構造を作る
- そして、その束となったトランザクションをサブタイプで表現する



```
trait NoTransaction  
  
trait ReadSlave1 extends NoTransaction  
  
trait ReadSlave2 extends NoTransaction  
  
trait ReadMaster  
  extends ReadSlave1 with ReadSlave2  
  
trait Write extends ReadMaster
```

- サブタイプ関係を使ってトランザクションの束構造をプログラム内に**型レベル**でエンコードできた

FUJITASK と合成

- **FUJITASK** の合成演算である `flatMap` は次のようになっている

```
trait Task[-R, +A] {  
  def flatMap[ER <: R, B](f: A => Task[ER, B]): Task[ER, B]  
}
```

- これはたとえば次のようになる
 - ① `Task[ReadSlave1, A]` と `Task[ReadSlave1, B]` を合成したら、`Task[ReadSlave1, B]` となる
 - ② `Task[ReadSlave1, A]` と `Task[ReadSlave2, B]` を合成したら、`Task[ReadMaster, B]` となる
 - ③ `Task[ReadSlave1, A]` と `Task[ReadMaster, B]` を合成したら、`Task[ReadMaster, B]` となる

FUJITASK の実行

- **FUJITASK** はモナドなので、for式でまとめていくことができる

```
val transaction = for {  
  _ <- paymentRepository.update(payment)  
  _ <- userPermissionRepository.update(userPermission)  
  _ <- coachRepository.assign(coach, user)  
} yield ()  
  
transaction.run()
```

- モナド構文ですっきり書ける
- オブジェクト指向プログラミングのサブタイプ、関数型プログラミングの型クラス・モナドが美しく融合している
 - サブタイプがない Haskell、モナド構文・型クラスがない Java に **FUJITASK** は実装できない

Extensible Effects

- モナドは1つの効果*を抽象化する1つの手段
- 1つのモナドは1つの効果しか抽象化できないため、
`Reader[Env, Future[Either[Err, Option[A]]]]`
のようなモナドスタックを使って複数の効果を表現する
 - しかしこうすると for式で内側のモナドへアクセスしにくくなる 😞
- モナドトランスフォーマーはこのような問題を解決するが、あるモナドについて専用のモナドトランスフォーマーが必要 😞
- *Extensible Effects* は特別な仕組みなしにモナドスタックを表現可能 😊

*副作用のこと。最近だと“計算効果 (Computational effect)”や単に“効果 (Effect)”と言う。

まとめ

- トランザクションはしばしば束構造を成す
- 束構造を型レベルにエンコードする手段としてサブタイプが使える
- 束構造に基づくアドホックな処理を、サブタイプと型クラスで実現できる
- オブジェクト指向プログラミングと関数型プログラミングは美しく融合されうる

参考文献

- [1] 結城清太郎.
ドワンゴ秘伝のトランザクションモナドを解説！, 2015.

Thank you for your attention!