Fujitask meets Extensible Effects

Hikaru Yoshimura（吉村 優）

Recruit Markting Partners Co., Ltd.
yyu@mental.poker

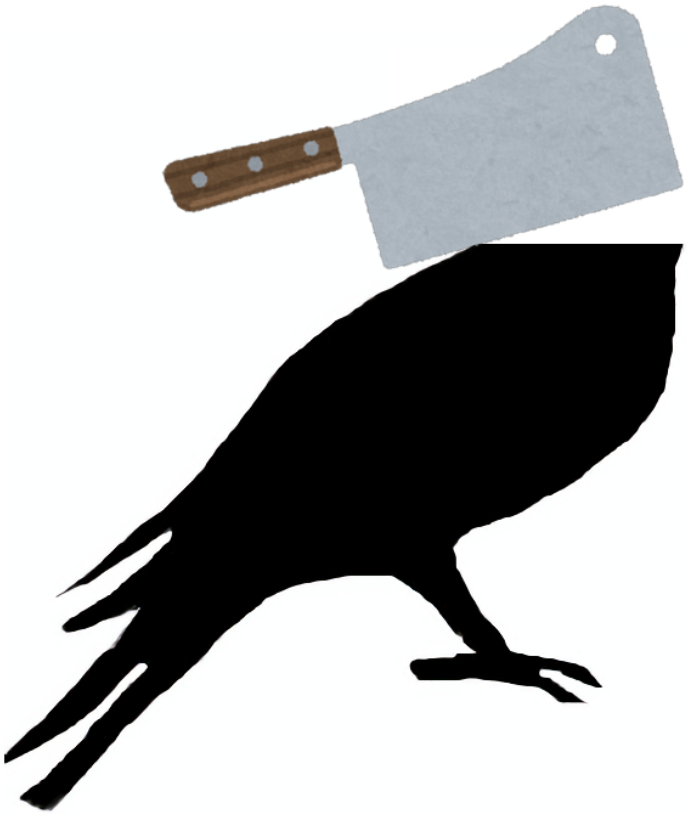
ScalaMatsuri on June 29, 2019

(y-yu/fujitask-eff-slide@622b076)

目次

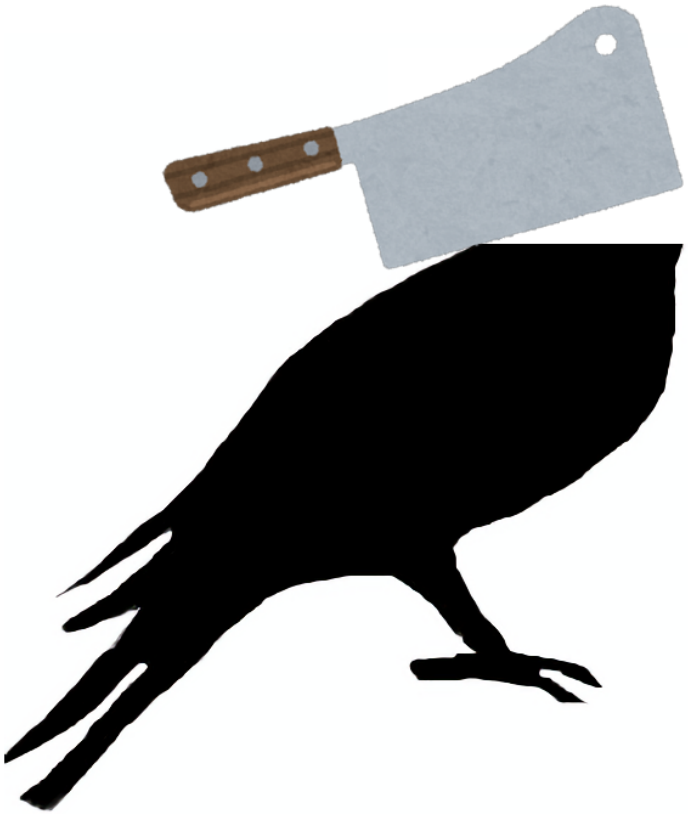
1. 自己紹介
2. トランザクションとは？
3. トランザクションのやり方
4. モナド版 **Fujitask**
   - **Fujitask** とサブタイプ
5. **Fujitask** と Extensible Effects
6. まとめ

自己紹介



Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

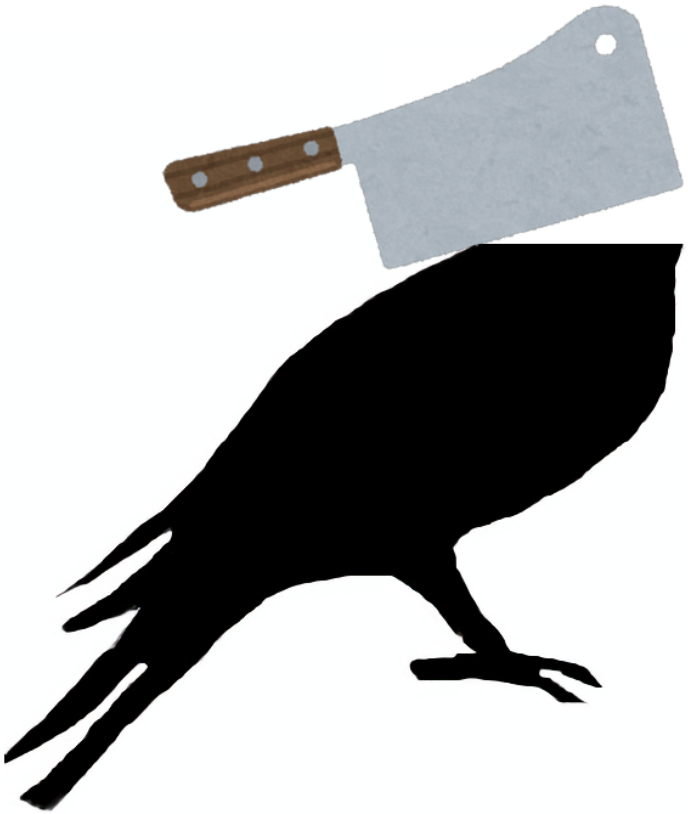
自己紹介



- 筑波大学 情報科学類卒（学士）

Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

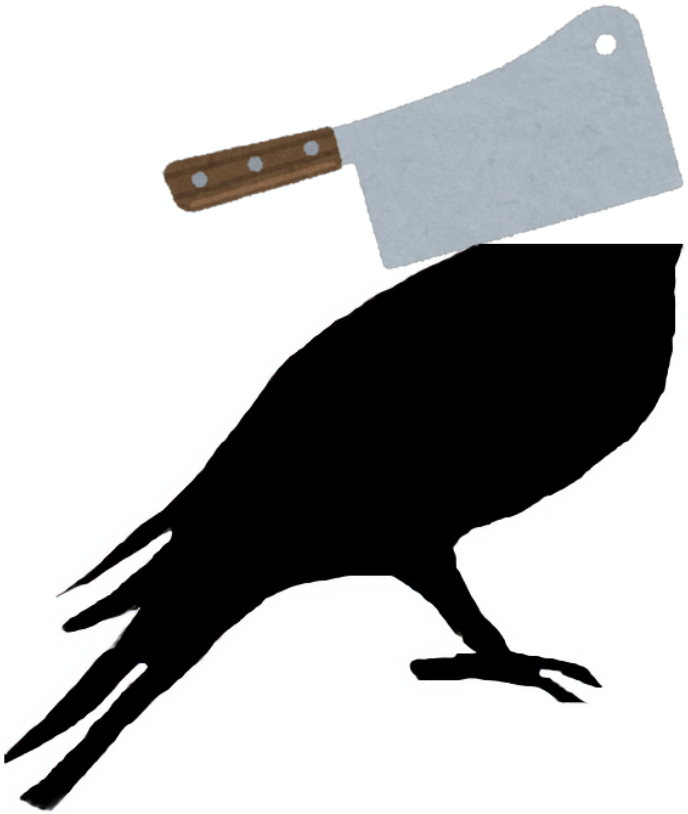
自己紹介



- 筑波大学 情報科学類卒（学士）
- プログラム論理研究室

Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

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
- LaTeX とか Scala とか

Twitter	@_yyu_
Qiita	yyu
GitHub	y-yu

トランザクションとは？

トランザクションとは？

- **トランザクション**は次を満す

トランザクションとは？

“

- **トランザクション**は次を満す
  - 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

”

トランザクションとは？

- **トランザクション**は次を満す

“

  - 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. コーチング機能が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

トランザクションとは？

- **トランザクション**は次を満す

“

  - 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. コーチング機能が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

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

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()
```

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

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

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

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）

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)
}
```

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

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

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？

トランザクションのやり方

- ❷ 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

トランザクションのやり方

- ❷ 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？

トランザクションのやり方

- ❷ 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 }
```

トランザクションのやり方

- ❷ 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ったとしても、新しく処理を書かなければならない

トランザクションのやり方

- ❷ 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ったとしても、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

トランザクションのやり方

- ❷ 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ったとしても、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）

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

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

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

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)
}
```

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！

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

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）

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）

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）

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

サブタイプと束

- Scala のサブタイプ関係は**束**を作る

サブタイプと束

- Scala のサブタイプ関係は**束**を作る

束

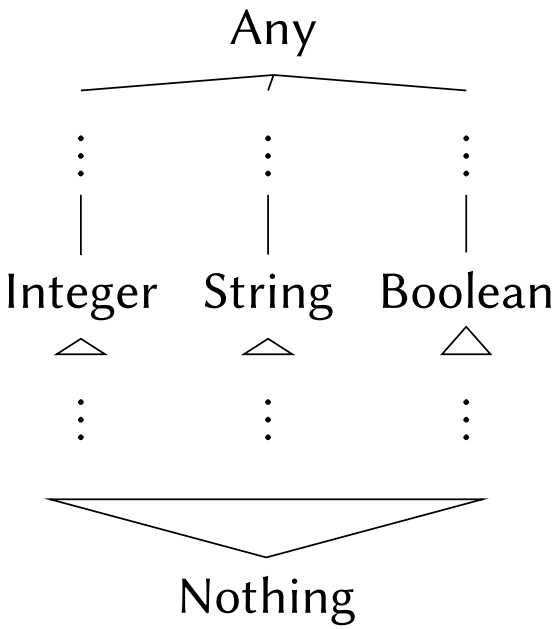
任意の 2 つの元が一意な上限および下限を持つ半順序集合のこと

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 つの元が一意な上限および下限を持つ半順序集合のこと

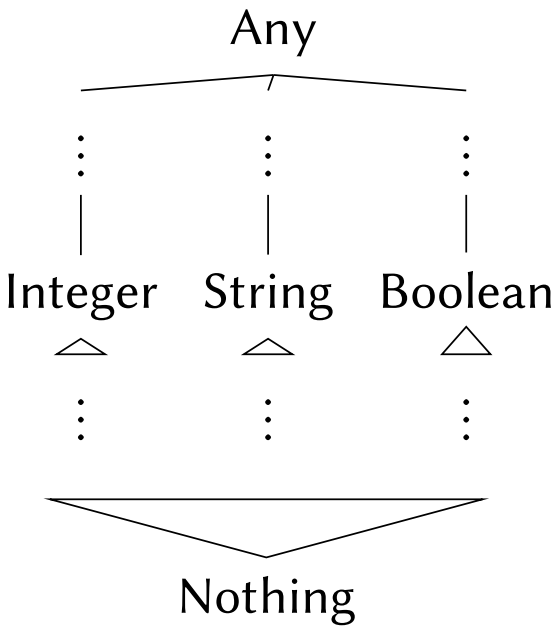


サブタイプと束

- 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



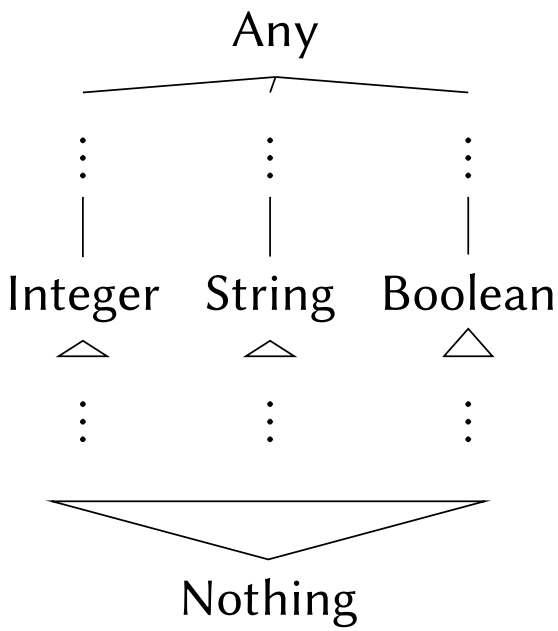
- どんな 2 つの元も、一意な共通の祖先と子孫をそれぞれ持つ

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



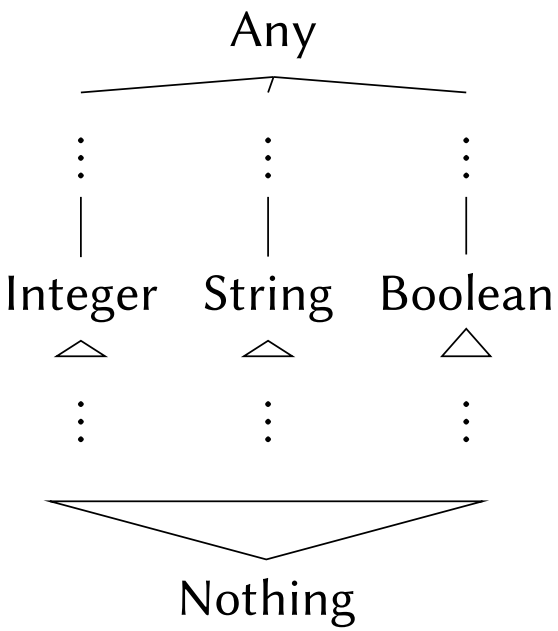
- どんな 2 つの元も、一意な共通の祖先と子孫をそれぞれ持つ
- 型システム理論においては、ボトム（`Nothing`）を入れると複雑になると知られており避けられる傾向にあるが、Scala には入っている

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

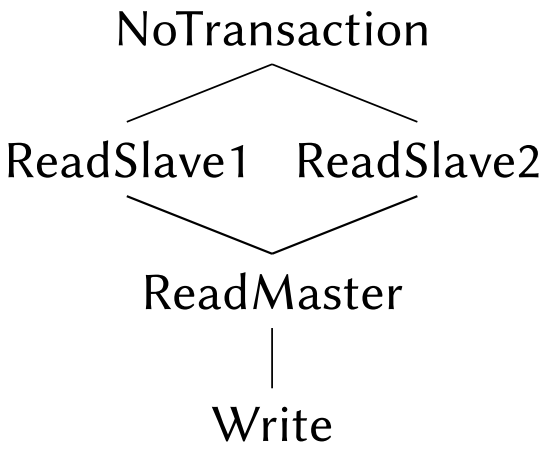
- 実はトランザクションは束構造を作る

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

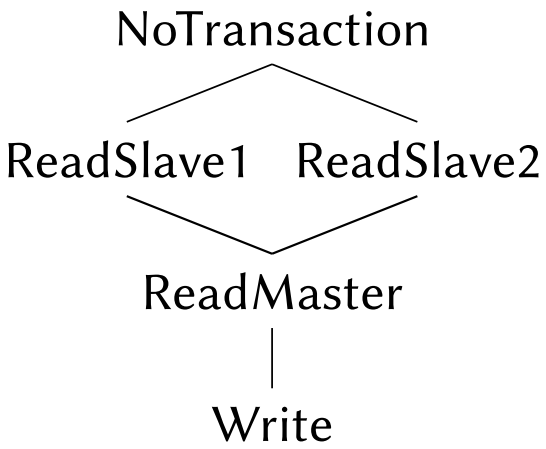
トランザクションと束

- 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



トランザクションと束

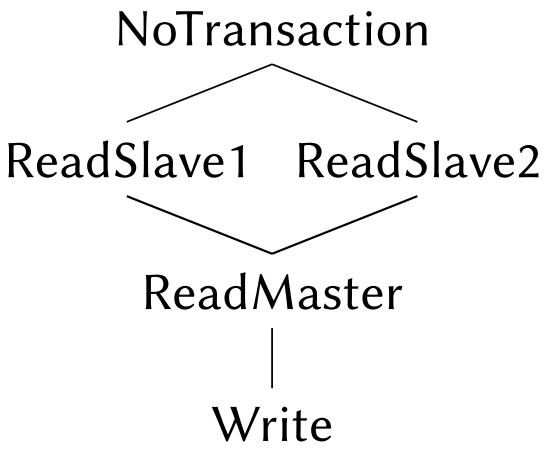
- 実はトランザクションは束構造を作る
- そして、その束となったトランザクションをサブタイプで表現する




```
trait NoTransaction

trait ReadSlave1 extends NoTransaction

trait ReadSlave2 extends NoTransaction

trait ReadMaster
  extends ReadSlave1 with ReadSlave2

trait Write extends ReadMaster
```

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と合成

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

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]
}
```

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

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

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

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

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.assgin(coach, user)
} yield ()

transaction.run()
```

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.assgin(coach, user)
} yield ()

transaction.run()
```

- モナド構文ですっきり書ける

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.assgin(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

ここまでのまとめ

ここまでのまとめ

- トランザクションはしばしば束構造を成す

ここまでのまとめ

- 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

ここまでのまとめ

- 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

ここまでのまとめ

- 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
- オブジョ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

Extensible Effects

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

Extensible Effects

- **モナド**は 1 つの**効果***を抽象化する 1 つの手段

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

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

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

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😞

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

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😞

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

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）” と言うこともある。

Fujitask と Extensible Effects

- **Fujitask** はサブタイプが必要

Fujitask と Extensible Effects

- **Fujitask** はサブタイプが必要
- Extensible Effects は Haskell 生まれ、でもサブタイプはない！

Fujitask と Extensible Effects

- **Fujitask** はサブタイプが必要
- Extensible Effects は Haskell 生まれ、でもサブタイプはない！

Fujitask を Extensible Effects へ持っていけるか？

Fujitask と Extensible Effects

- **Fujitask** はサブタイプが必要
- Extensible Effects は Haskell 生まれ、でもサブタイプはない！

Fujitask を Extensible Effects へ持っていけるか？

- ねこはる（`@halcat0x15a`）さんの作った kits-eff を利用
  - atnos-eff とは違ったサブタイプを利用した Extensible Effects の実装

Fujitask と Extensible Effects

- **Fujitask** はサブタイプが必要
- Extensible Effects は Haskell 生まれ、でもサブタイプはない！

Fujitask を Extensible Effects へ持っていけるか？

- ねこはる（`@halcat0x15a`）さんの作った kits-eff を利用
  - atnos-eff とは違ったサブタイプを利用した Extensible Effects の実装
- できたものはここ👇
  `https://github.com/y-yu/fujitask-eff`

Example

- こういう感じでいろいろな効果と一緒に forに入れられる

```
case class User(id: Long, name: String)
// create table `user` (
//   `id` bigint not null auto_increment,
//   `name` varchar(64) not null
// )

val eff = for {
  name <- Reader.ask[String]
  user  <- userRepository.create(name)
} yield {
  logger.info(s"user is \$user")
}
Fujitask.run(Reader.run("piyo")(eff))
```

```
ReadWriteRunner begin --------->
user is Some(User(2,piyo))
<--------- ReadWriteRunner end
```

Example

- こういう感じでいろいろな効果と一緒に forに入れられる

```
case class User(id: Long, name: String)
// create table `user` (
//   `id` bigint not null auto_increment,
//   `name` varchar(64) not null
// )

val eff = for {
  name <- Reader.ask[String]
  user  <- userRepository.create(name)
} yield {
  logger.info(s"user is \$user")
}
Fujitask.run(Reader.run("piyo")(eff))
```

```
ReadWriteRunner begin --------->
user is Some(User(2,piyo))
<--------- ReadWriteRunner end
```

- デモ

Example

- こういう感じでいろいろな効果と一緒に forに入れられる

```
case class User(id: Long, name: String)
// create table `user` (
//   `id` bigint not null auto_increment,
//   `name` varchar(64) not null
// )

val eff = for {
  name <- Reader.ask[String]
  user  <- userRepository.create(name)
} yield {
  logger.info(s"user is \$user")
}
Fujitask.run(Reader.run("piyo")(eff))
```

```
ReadWriteRunner begin --------->
user is Some(User(2,piyo))
<--------- ReadWriteRunner end
```

- デモ
- 今回はインタープリターの flatMapだけ見てみましょう

Fujitaskのインタープリター

```
object Fujitask {
  def run[I <: Transaction: Manifest, A](
    eff: Eff[I, A]
  )(
    implicit runner: FujitaskRunner[I],
    ec: ExecutionContext
  ): Future[A] = {
    def handle(i: I) = new ApplicativeInterpreter[Fujitask, Any] {
      def flatMap[T, B](fa: Fujitask with Fx[T])(k: T => Eff[Any,
    Future[B]]): Eff[Any, Future[B]] =
        fa match {
          case Execute(f) =>
            Eff.Pure(f(ec).flatMap(a => Eff.run(k(a))))
          case _: Ask[I] =>
            k(i.asInstanceOf[T])
        }
    }
}
```

- 😕

Fujitaskのインタープリター

- 解説しようと思ったものの、ちょっと時間がない……😇

Fujitaskのインタープリター

- 解説しようと思ったものの、ちょっと時間がない……😇
- 今回はモチベーションだけで勘弁してください🤷

Fujitaskのインタープリター

- 解説しようと思ったものの、ちょっと時間がない……😇
- 今回はモチベーションだけで勘弁してください🤷
- 解説資料 [1] があるので興味がでた人はこれを読んで！

Fujitaskのインタープリター

- 解説しようと思ったものの、ちょっと時間がない……😇
- 今回はモチベーションだけで勘弁してください🤷
- 解説資料 [1] があるので興味がでた人はこれを読んで！
- あと、kits-eff は [2] を読むといいです

まとめ

まとめ

- **Fujitask** は関数型プログラミングとオブジェクト指向プログラミングのよい融合

参考文献

[1] 吉村優.
Extensible Effects でトランザクションモナド “Fujitask” を作る, 2019.

[2] @halcat0x15a.
Scala らしい Eff を目指して.
進捗大陸 05, 第 6 章. 4 2019.
https://shinchokutairiku.booth.pm/items/1309694.

[3] 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!