実践モナド

JAIST D1 宇田 拓馬

自己紹介

- 名前: 宇田拓馬
- 所属: JAIST 青木研究室 D2
- 研究キーワード: 形式仕様記述, 定理証明支援
- 趣味: 読書, プログラミング
- Twitter: @hennin_ltn

モナドとは

こんなやつ

```
permutation :: List a -> List (a, a)
permutation xs = do
  x1 <- xs
  x2 <- xs
  pure $ [x1, x2]
```

何がうれしい？

- do記法が使える！！！！！
- あと抽象化できる
  → 色々な概念を同じ操作で扱える

Listの例

```
permutation :: List a -> List (a, a)
permutation xs = do
  x1 <- xs
  x2 <- xs
  pure $ [x1, x2]
```

何がうれしい？

- do記法が使える！！！！！
- あと抽象化できる
  →色々な概念を同じ操作で扱える

Maybeの例

```
head :: List a -> Maybe a
tail :: List a -> Maybe a

second :: List a -> Maybe a
second xs = do
  xs' <- tail xs
  pure $ head xs'
```

何がうれしい？

- do記法が使える！！！！！
- あと抽象化できる
  →色々な概念を同じ操作で扱える

Stateの例

```
fact :: Int -> Int -> Int
fact acc 0 = acc
fact acc n = fact_acc (n * acc) (n - 1)

factS :: Int -> State Int Int
factS n = do
  acc <- get
  put (acc * n)
  fact_acc (n - 1)
```

前提知識

カインドと高カインド型

カインドとは？

- 型の型
- 型のarity
- arityが2以上の型を高カインド型という

```
1 :: Int
suc :: Int -> Int
conj :: Boolean -> Boolean -> Boolean

Int :: *
List :: * -> *
Tuple :: * -> * -> *
```

型クラス

- Interfaceに似たようなやつ
- Addable型クラスのインスタンスには関数addが実装される
- NはAddableのインスタンス

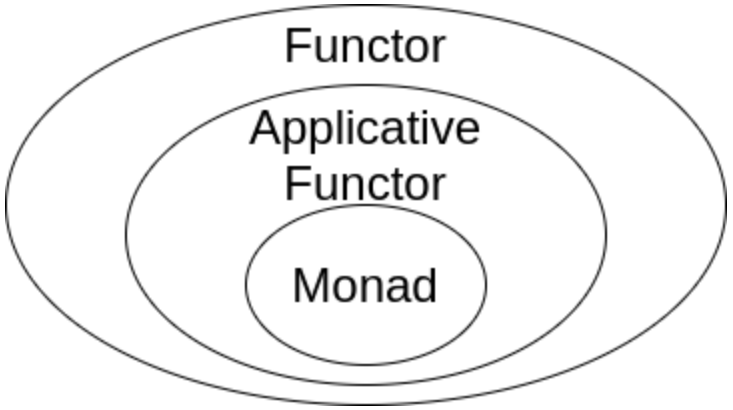
```
class Addable a where
  add :: a -> a -> a

data N = Zero | Suc N

instance Addable N where
  add n Zero = n
  add n (Succ m) = add (Succ n) m
```

モナドとは？

- 型引数を1つ受け取る高カインド型に対する型クラス
  - List
  - Maybe
  - State s
- アプリカティブファンクターは後から見つかった



ファンクター

- 型引数を1つ受け取る高カインド型に対する型クラス
- ファンクター `f` には以下の関数が実装されている
  - `map :: (a -> b) -> f a -> f b`
- `map` は以下を満たす (ファンクター則)
  - Identity: `map identity = identity`
  - Composition: `map (f . g) = map f . map g`
- `map` のイメージとしては「中の値に関数を適用する」
  - `map :: (a -> b) -> (f a -> f b)`

```
class Functor f where
  map :: (a -> b) -> f a -> f b
```

ファンクター

- `List a`は「`a`の値を集めたもの」
- Listの`map`は「Listに含まれるすべての値に`f`を適用する」

```
class Functor f where
  map :: (a -> b) -> f a -> f b

data List a = Nil | Cons a

instance Functor List where
  map f (Cons x xs) = Cons (f x) xs
```

```
> map (* 2) [1, 2, 3]
[2, 4, 6]
```

ファンクター

- `Maybe a`は「`a`の値であるか、そうでないもの」
- Maybeの`map`は「値がある場合にはその値に`f`を適用する」

```
class Functor f where
  map :: (a -> b) -> f a -> f b

data Maybe a = Nothing | Just a

instance Functor Maybe where
  map f Nothing = Nothing
  map f (Just x) = Just (f x)
```

```
head :: List a -> Maybe a
head Nil = Nothing
head (Cons x xs) = Just x

map (* 7) (head [1, 2, 3])
> Just 7
```

ファンクター

- `State s a`は「状態`s`を持ち、`a`の値を返す計算(関数)」
- Stateの`map`は「計算結果に`f`を適用する」

```
class Functor f where
  map :: (a -> b) -> f a -> f b

data State s a = State (s -> (a, s))

instance Functor (State s) where
  map f (State a) = State (\s -> map (\(b, s) -> (f b, s')) (a s))
```

モナド

- 型引数を1つ受け取る高カインド型に対する型クラス
- モナドはファンクターである
- モナド `m` には以下の関数が実装されている
  - `map :: (a -> b) -> m a -> m b`
  - `pure :: a -> m a`
  - `bind :: m a -> (a -> m b) -> m b`
- `map`, `pure`, `bind` は以下を満たす (モナド則)
  - Left Identity: `bind (pure x) f = f x`
  - Right Identity: `bind x pure = x`
  - Associativity `bind (bind x f) g = bind x (\k -> (bind (f k) g))`

```
class (Functor m) <= Monad m where
  pure :: a -> m a
  bind :: m a -> (a -> m b) -> m b
```

モナド

- `pure` のイメージとしては「単一の値をくるむ」
- `bind` のイメージとしては「`m a` から `a` を取り出して `a -> m b` を適用する」
  - 本当はモナドに取り出す操作はないので, 適用結果を畳み込んでいる (`m (m b) -> m b`)

```
class (Functor m) <= Monad m where
  pure :: a -> m a
  bind :: m a -> (a -> m b) -> m b
```

モナド

- モナドは`bind`を使って処理を合成できる

```
data Maybe a = Nothing | Just a

head :: List a -> Maybe a
head Nil = Nothing
head (Cons x xs) = Just x

tail :: List a -> Maybe (List a)
tail Nil = Nothing
tail (Cons x xs) = Just xs

second :: List a -> Maybe a
second xs = bind (tail xs) head
```

do記法

- do記法は`bind`のシンタックスシュガー

```
second :: List a -> Maybe a
second xs = bind (tail xs) head

second :: List a -> Maybe a
second xs = do
  xs' <- tail xs
  head xs'
```

do記法

- do記法がないとかなり複雑な記述となる
- `bind :: m a -> (a -> m b) -> m b`は「`m a`から`a`を取り出して`a -> m b`を適用する」
- `pure :: a -> m a`は「`a`をモナド`m`でくるむ」
- `<-`は「`m a`から`a`を取り出す」と読める

```
last :: List a -> Maybe a
last Nil = Nothing
last (Cons x Nil) = Just x
last (Cons x xs) = last xs

headLast :: List a -> (a, a)
headLast = bind (head xs) (\h -> bind (tail xs) (\l -> (h, l)))

headLast :: List a -> (a, a)
headLast xs = do
  h <- head xs
  l <- last xs
  pure $ (h, l)
```

- do記法がないとかなり複雑な記述となる
- `bind :: m a -> (a -> m b) -> m b`は「`m a`から`a`を取り出して`a -> m b`を適用する」
- `pure :: a -> m a`は「`a`をモナド`m`でくるむ」
- `<-`は「`m a`から`a`を取り出す」と読める

```haskell
permutation :: List a -> List (a, a)
permutation xs = do
  x1 <- xs
  x2 <- xs
  pure $ [x1, x2]
```

```haskell
fact :: Int -> Int -> Int
fact acc 0 = acc
fact acc n = fact_acc (n * acc) (n - 1)

factS :: Int -> State Int Int
factS n = do
  acc <- get
  put (acc * n)
```