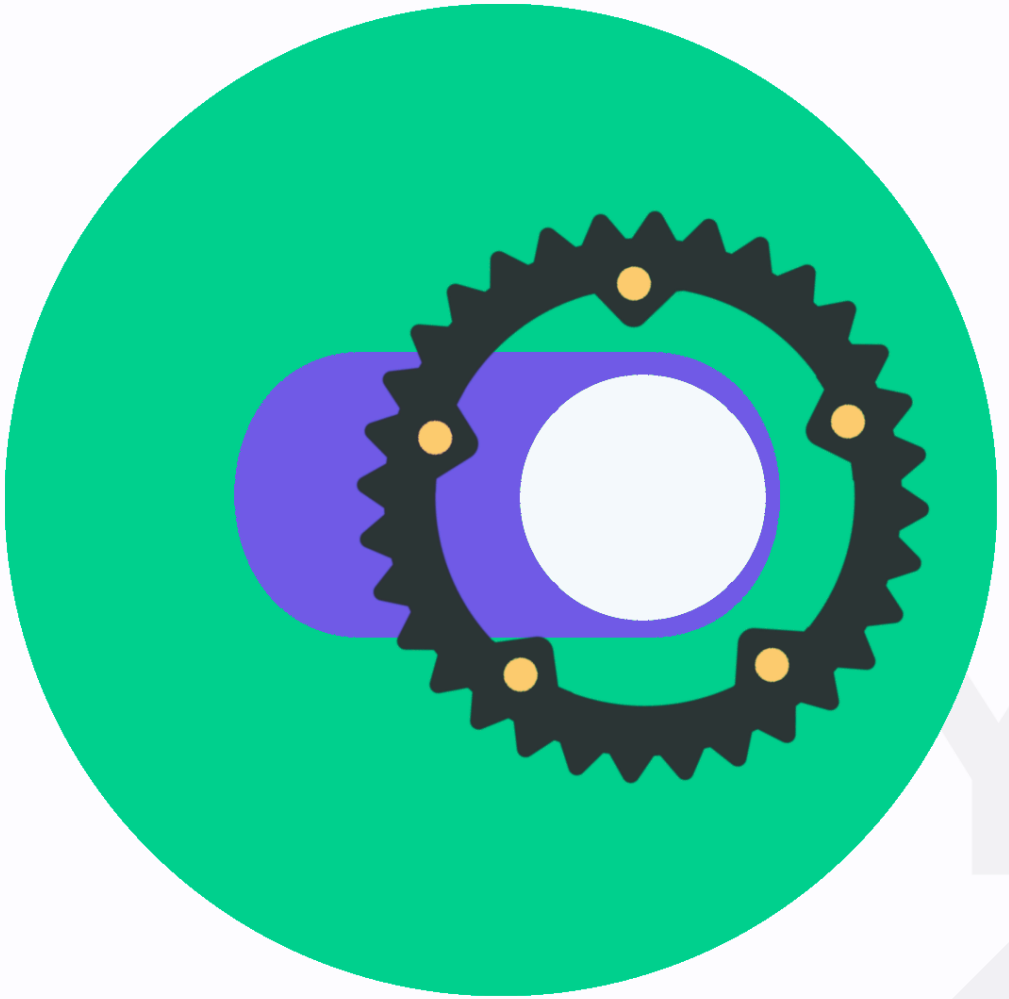
Haskellの型クラス

@coord_e

おはようございます！！！
coord_eです、よろしくどうぞ。



- coins20 AC
- Twitter
  - @coord_e
  - @coord2e
- プログラミング言語の理論と実装に興味がある

Haskell...?

こういう感じの関数型言語です（省略）

- 関数適用: `f x`
- 関数定義: `f x = e`
- 型アノテーション: `x :: t`

型...?

ここでは式を分類します（省略）

型クラスっていうのがあるらしいです

話すこと

- 型クラスの**型推論**の実装イメージ
- 型クラスによる**オーバーロード**の実装イメージ

型クラスがやること

1. 型を分類することで抽象化
2. 名前がぶら下がっているので名前と実装の分離ができる
    - オーバーロードとも
    - 型クラスと名前、型と実装が紐ついている

例

```haskell
class Eq a where
  eq :: a -> a -> Bool

-- 同値比較ができる型のリストの上でのみ行える操作
member :: Eq a => a -> [a] -> Bool
member x [] = False
member x (h : t) | eq x h    = True
                 | otherwise = member x t
```

- `Eq a` が `a` の上で `eq` が使えることを示している
- `Eq` は `eq` が定義された型の集合とみれる
- ただし、`eq` の実装は型によって異なって良い

クラスとインスタンス

※OOPはね、関係ない

```
class Eq a where
  eq :: a -> a -> Bool

instance Eq Char where
  eq = ...  -- プリミティヴな操作...

instance Eq Int where
  eq = ...  -- プリミティヴな操作...

instance Eq Bool where
  eq True  True  = True
  eq False False = True
  eq _     _     = False
```

実装

- 制約があったときに、
  マッチするインスタンス宣言があったら制約を解消
- 例:

```
member 'a' ['b', 'c'] :: Eq Char => Bool
```

  - `instance Eq Char where ...` のインスタンス宣言が存在するので、 `Eq Char` を解消
  - `Bool` に型付け

だけじゃない

```
instance (Eq a, Eq b) => Eq (a, b) where
  eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2
```

- `(Eq a, Eq b)` なときに限り、 `Eq (a, b)`
- 定義中で `a` `b` に対する
  `eq` が使われていることに注意

実装

- `C t` の制約があったときに、
  マッチするインスタンス宣言があったら `C t` を
  マッチするインスタンス宣言の制約と置き換える
- 例:

```
eq (True, 'a') (False, 'b') :: Eq (Bool, Char) => Bool
```

1. `instance (Eq a, Eq b) => Eq (a, b)` より、
   `Eq (Bool, Char)` → `Eq Bool, Eq Char`
2. `instance Eq Bool` が存在するので解消
3. `instance Eq Char` が存在するので解消

クラス制約

```
class Eq a => Ord a where
  lt :: a -> a -> Bool
  gt :: a -> a -> Bool
  ...
```

- `Ord a` が `Eq a, Ord a` と同じ意味になる
- `Ord t` のインスタンス宣言時に `Eq t` でないとエラー
- `Eq a, Ord a` と制約があったときに `Ord a` にまとめて見た目を綺麗にできる

はい

- こんな方法でHMに手を加えれば型推論はできる
  - Jones, Mark P. "Typing haskell in haskell." Haskell workshop. Vol. 7. 1999.
- 実際はHNF（Head-Normal Form）に到達するまで繰り返すので、停止するんですか？という話があるが…
  - 停止するんですか議論はよくわかっとらんスマン
  - ナントカconditionというのがあります

実装の選択の実装

型クラスがやること

1. 型を分類することで抽象化
2. **名前がぶら下がっているので名前と実装の分離ができる（オーバーロード）**

実装の選択

- 型クラスに必要な実装が入ったオブジェクトを **辞書** と呼ぶことにする
  - 辞書からメソッドを取り出すことができる
- それぞれのインスタンスには実装が入った対応する辞書が存在する
- 型制約を伴った名前はそれぞれの型制約の型クラスに対応する辞書を受け取ることにする
- 型推論の過程で辞書をうまく受け渡すことで実装の選択が行われる

実装

- `C t` の制約があったときに、
  マッチするインスタンス宣言があったら `C t` を
  マッチするインスタンス宣言の制約と置き換え、
  制約を発生させた名前に対応する辞書を渡す

これから例をお見せします

- `x@d` で名前 `x` に辞書 `d` を渡すことにします
- `x@d = ...` で辞書 `d` を受け取る名前 `x` を定義することにします
- ただの関数適用で実装しても特に問題ない

定義側の例

```
member x [] = False
member x (h : t) | eq x h    = True
                 | otherwise = member x t
```

1. `eq` から `Eq a` が登場し下のように型が付く（省略）

```
member :: Eq a => a -> a -> Bool
```

定義側の例

```
member@dict x [] = False
member@dict x (h : t) | eq@dict x h = True
                      | otherwise   = member x t
```

2. `Eq a` にマッチするインスタンスはなく、受け取った辞書をそのまま使うようにする
3. 汎化

```
member :: forall a. Eq a => a -> a -> Bool
```

使用側の例

```
member (True, 'a') [] :: Eq (Bool, Char) => Bool
```

使用側の例

```
-- `eqPair` は `Eq (a, b)` の実装が入った辞書
member@eqPair (True, 'a') [] :: (Eq Bool, Eq Char) => Bool
```

1. `instance (Eq a, Eq b) => Eq (a, b)` より、`Eq (Bool, Char)` → `Eq Bool, Eq Char`
   - `Eq (a, b)` の発生源は `member` なので、`member` に辞書を渡す
   - ここで `eqPair` は `Eq Bool` と `Eq Char` の実装を必要としていることに注意

使用側の例

```
-- `eqBool` は `Eq Bool` の実装が入った辞書
member@(eqPair@eqBool) (True, 'a') [] :: Eq Char => Bool
```

2. `instance Eq Bool` が存在するので解消
   - `Eq Bool` の発生源は `eqPair` なので、`eqPair` に辞書を渡す

使用側の例

```
-- `eqChar` は `Eq Char` の実装が入った辞書
member@(eqPair@eqBool@eqChar) (True, 'a') [] :: Bool
```

3. `instance Eq Char` が存在するので解消
   - `Eq Char` の発生源は `eqPair` なので、`eqPair` に辞書を渡す

はい

- 型推論の過程で辞書をうまく受け渡すことで実装できる
- Dictionary Passingについて...
  - Demystifying Type Classes - http://okmij.org/ftp/Computation/typeclass.html
  - Wadler, Philip, and Stephen Blott. "How to make ad-hoc polymorphism less ad hoc." Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM, 1989.

高階型クラスについて

高階型クラス

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

instance Functor [] where
  fmap = map

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

高階型クラス

- Jones, Mark P. "A system of constructor classes: overloading and implicit higher-order polymorphism." Journal of functional programming 5.1 (1995): 1-35.
- カインド付け
  - これはやるだけ
- 変換後のプログラムに型を付けづらいと思っている
  - Rank2多相が必要なんじゃないかなあ
  - *別に高階型クラスに限った話ではないが、問題になりやすい*

高階型クラスとRank2多相

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

void :: Functor f => f a -> f ()
void = fmap (const ())
```

↓

```haskell
data FunctorDict f
  = FunctorDict
  { fmap :: forall a b. (a -> b) -> f a -> f b
  }

void :: FunctorDict f -> f a -> f ()
void dict = fmap dict (const ())
```

まとめ

話したこと

- 型クラスというのがあって、名前と実装が分離できる
- HM型推論にチョコっと手を加えることで
  型推論の過程で変換を行い実装できる
- 高階型クラスというのがある

展望

- 高階型クラス含め実装したい
  - 変換先の言語として手軽なものがなくてしんどいよ〜
- CHR (Constraint Handling Rules) との関連に興味ある
  - Glynn, Kevin, Martin Sulzmann, and Peter J. Stuckey. "Type classes and constraint handling rules." arXiv preprint cs/0006034 (2000).
  - Alves, Sandra, and Mário Florido. "Type inference using constraint handling rules." Electronic Notes in Theoretical Computer Science 64 (2002): 1-17.

ありがとうございました

👇スライド👇

coorde-slide-type-class.netlify.app