

関数・論理型プログラミング実験 Haskell演習第2回 (通算第9回)

松田 一孝
TA: 武田広太郎 寺尾拓

今日の内容

- 型クラスをより詳しく

復習：型クラス

- 型に可能な操作を抽象化
 - ◆ 正確には型でなく型構成子でよい

例

型クラスEq
の宣言

```
class Eq a where  
    (==) :: a -> a -> Bool
```

「メソッド」と呼ばれる

Eq Boolの
インスタンス
宣言

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

Eqのメソッドを全て定義

復習：型クラスの利用

```
-- Eqのメソッドを使ったプログラム
allEqual []      = True
allEqual (x:xs)  = go x xs
  where
    go y []      = True
    go y (z:zs) = (y==z) && go z zs
```

```
*Main> :t allEqual
Eq t => [t] -> Bool
```

```
*Main> allEqual [True, False]
False
```

Eqのインスタンスtについて
[t]に適用するとBoolを返す

BoolはEqのインスタンス
なので[Bool]に適用できる

Mission

- 大小関係が比較可能な型を表す
型クラスを定義せよ

一つの解

```
class Ord a where
  (<=) :: a -> a -> Bool

x >= y = y <= x
x > y = not (x <= y)
x < y = not (x >= y)
min x y = if x <= y then x else y
max x y = if x >= y then x else y
```

問題点

- 部分型クラス関係を反映できていない

```
minimum :: Ord a => [a] -> a
minimum = foldr1 min
maximum :: Ord a => [a] -> a
maximum = foldr1 max

allEqual' :: Ord a => [a] -> Bool
allEqual' xs = minimum xs == maximum xs
```

型エラー：
aがEqに属していない

改良版

Ordのインスタンスは
すべてEqのインスタンス

```
class Eq a => Ord a where  
  (<=) :: a -> a -> Bool  
  
...
```

```
allEqual' xs = minimum xs == maximum xs
```

```
*Main> :t allEqual'  
Ord t => [t] -> Bool
```


インスタンス宣言

- tがOrdのインスタンスであるためにはEqのインスタンスでなければならない
 - ◆ Ord tのインスタンス宣言を行うにはEq tのインスタンス宣言が必要

```
data T = A | B
```

```
instance Eq T where
    A == A = True
    B == B = True
    _ == _ = False
```

```
instance Ord T where
    A <= B = True
    B <= B = True
    _ <= _ = False
```

さらなる改良

- $<=$ や $=$ 以外の演算を定義することで
EqやOrdのインスタンスを宣言したい
 - ◆ $/=$ の定義のほうが楽な場合
 - ◆ 効率の問題

デフォルト実装

```
class Eq a where
  (==) :: a -> a -> Bool
  x == y = not (x /= y)
  (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

(==) と (/=) の
いづれかを実装すれば
もう一方も定義される

```
data T = A | B
```

```
instance Eq T where
  A == A = True
  B == B = True
  _ == _ = False
```

OK

```
instance Eq T where
  A /= A = False
  B /= B = False
  _ /= _ = True
```

OK

注意

- どのメソッドを定義すればよいかはデフォルト実装による

どれでもOK

```
class Eq a => Ord a where
  (<=) :: a -> a -> Bool
  x <= y = y > x || x == y
  (<)  :: a -> a -> Bool
  x < y = not (y <= x)
  (>=) :: a -> a -> Bool
  x >= y = not (y < x)
  (>)  :: a -> a -> Bool
  x > y = not (y >= x)
```

<=か>を定義する必要

```
class Eq a => Ord a where
  (<=) :: a -> a -> Bool
  x <= y = y > x || x == y
  (<)  :: a -> a -> Bool
  x < y = not (y <= x)
  (>=) :: a -> a -> Bool
  x >= y = y <= x
  (>)  :: a -> a -> Bool
  x > y = not (x <= y)
```

Mission

- f a の全ての a 要素に関数を適用できる
型構成子 f を表す型クラスを定義せよ
 - ◆ 型構成子：型から別の型を作るもの
 - * 例：以下の`[]`や`Bin`

```
map :: (a -> b) -> [a] -> [b]
```

```
data Bin a = Leaf | Node a (Bin a) (Bin a)
mapBin :: (a -> b) -> Bin a -> Bin b
mapBin f Leaf = Leaf
mapBin f (Bin a x y) =
  Node (f a) (mapBin f x) (mapBin f y)
```

型クラスFunctor

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

リストの場合

```
instance Functor [] where  
  fmap = map
```

二分木の場合

```
instance Functor Bin where  
  fmap = mapBin
```

利点

- 「要素」についての統一的な操作
 - ◆ 標準ライブラリ中の
Functorのインスタンス (抜粋)

```
Functor []  
Functor Maybe  
Functor ((->) r)    -- 関数の返り値の部分  
Functor (Map k)     -- 連想配列の値の部分  
Functor Seq         -- 列  
Functor Tree        -- 薔薇木  
Functor ReadP       -- 構文解析コンビネータの結果
```

Mission

- コンテナを統一的に扱いたい
 - ◆ コンテナ：
ここでは全要素と
「形」に分離できるもの

一つの解

```
class Functor f => Container f where
  contents :: f a -> [a]
  fill :: f b -> [a] -> f a
```

ただし，以下を満たす

$\text{fill (fmap g t) (contents t)} = \text{t}$

もし $\text{length xs} = \text{length (contents t)}$ ならば
 $\text{contents (fill t xs)} = \text{xs}$

注意：上はGHCのライブラリにはないが，
Data.Traversableと実質的に同じ

インスタンス

```
instance Container [] where
  contents = id
  fill _ x = x
```

```
instance Container Bin where
  contents Leaf = []
  contents (Node a x y) = a:contents x ++ contents y
  fill t as = fst (go t as)
    where go Leaf as = (Leaf, as)
          go (Node _ x y) (a:as) =
            let (x', asx) = go x as
                (y', asy) = go y asx
            in (Node a x' y', asy)
```

利点

- 様々な操作を統一的に
 - ◆ 例：foldrの（ひとつの）一般化

```
fold :: Container t => (a -> b -> b) -> b -> t a -> b
fold f e t = foldr f e (contents t)
```

* 様々な演算

- 例：最小値の計算

```
minimum :: (Container t, Ord a) => t a -> a
minimum t = fold min 0 t
```

他の例

- 要素のナンバリング
 - ◆ 要素に一意的なIDをふる

```
numbering :: Container t => t a -> t (a, Int)
numbering t = fill t (go (contents t) 0)
  where
    go [] _ = []
    go (x:xs) n = (x, n) : go xs (n+1)
```

- 一般的なzip

```
zip' :: Container t => t a -> t b -> t (a, b)
zip' x y = fill x (zip (contents x)
                      (contents y))
```

まとめ

- 型クラスの利点
 - ◆ 異なる型（や型構成子）の操作に
統一的なインタフェースを与える

注意

- 型クラスの宣言には様々な制約がある
 - ◆ \Rightarrow の左に出てくるのは
型クラス 型変数
の形
 - ◆ \Rightarrow の右に出てくるのは
型クラス (型構成子 型変数 ... 型変数)
の型 (型変数は互いに異なる必要有)
- GHCには制約を緩める様々な拡張
 - ◆ FlexibleInstances, FlexibleContexts,
UndecidableInstances, MultiParamTypeClasses, ...

次回

- MonadやApplicativeのような「作用」を表す型クラス
- HaskellにおけるI/O等の扱い

第8回レポート課題
締切 6/24 13:00

問1

- 様々な型に対し，その型をEqやOrd，Showのインスタンスにしてみよ
 - ◆ ここでEqやOrdはPreludeのもの
 - * 資料にあるものではない
 - ◆ derivingは使わない
 - ◆ 対象とする型は型変数を持つものが望ましい
 - ◆ 少なくとも3つの型について行え

問2

- 様々な型構成子に対し，その型を Container のインスタンスにしてみよ
 - ◆ Container のメソッドを利用することで コンテナに統一的に適用できる関数を定義し，いろんな型のコンテナに適用してみよ
 - ◆ GHC に含まれる型構成子を対象にしてもよいが，Data.Traversable のインスタンスであることを利用してはいけない

問3 (1/3)

- 以下の型を準備する

```
newtype Fix f = In (f (Fix f))
```

- 以下の型を持つ関数を定義せよ

```
cata :: Functor f => (f a -> a) -> Fix f -> a
```

- ◆ ただし、以下を満たすように
cata f . In = f . fmap (cata f)

問3 (2/3)

- 型構成子 `NatF` と `ListF a` を以下で定める
 - ◆ `NatF` と `ListF b` を `Functor` のインスタンスにせよ

<pre>data NatF a = Zero Succ a data ListF b a = Nil Cons b a</pre>
--

- ◆ `Fix NatF` と `Fix (ListF b)` が何を表現しているか答えよ

問3 (3/3)

- 定義したcataを使い
以下の関数を実装せよ
 - ◆ `toList :: Fix (ListF a) -> [a]`
 - ◆ `toInt :: Fix NatF -> Int`
 - ◆ `add ::`
`Fix NatF -> Fix NatF -> Fix NatF`
 - ◆ `sum ::`
`Fix (ListF (Fix NatF)) -> Fix NatF`

補足

- newtypeは実装が同じ異なる型を定義
 - ◆ $\text{newtype } \text{型} \text{ 型変数} \dots \text{型変数}$
= 構成子 型
- ◆ dataとの主な違い
 - * $\text{newtype } T \ a = C \ a$ に対し,
 $C \ \perp$ は, \perp
 - * $\text{data } T \ a = C \ a$ に対し,
 $C \ \perp$ は, \perp ではない
 - \perp は, $\text{let } f = f \text{ in } f$ と等価な式

余談

- 正格な f に対し, 先の条件を満たす
正格な $\text{cata } f$ は唯一
 - ◆ 正格 : $f \perp$ が \perp と等価なこと
- 上記のような $\text{cata } f$ は
正格な g について
$$g . \text{cata } f = \text{cata } h$$
$$\text{if } g . f = h . (\text{fmap } g)$$
 - ◆ ループ融合

問4 (1/2)

○ 型クラス ListLike を以下で与える

◆ class ListLike f where

nil :: f a

cons :: a -> f a -> f a

app :: f a -> f a -> f a

toList :: f a -> [a]

fromList :: [a] -> f a

問4 (2/2)

- リストの末尾に要素を追加する関数
 $\text{snoc} :: \text{ListLike } f \Rightarrow f \ a \rightarrow a \rightarrow f \ a$
- リストを逆順にする関数
 $\text{rev} :: \text{ListLike } f \Rightarrow [a] \rightarrow f \ a$
を実装せよ
- nil , cons , app が $O(1)$ となる
インスタンスを与えよ
 - ◆ ただし, $\text{toList } (\text{fromList } x)$ が x と等しくなるように
- 上記インスタンスを利用し,
 $O(n)$ で動く reverse を書け

発展1 (オーバロードの悪夢)

- 次はUncyclopediaのHaskellの項にある式を，一部改変したものである
fix\$ (<\$>) <\$> (:) <*> ((<\$> (: [])) <\$>))
(=<<>) <\$> (*) <\$> (*2)) \$1
上記コードが何をするのか，どうして
そのような動作をするかを説明せよ
- ◆ Control.ApplicativeとData.Function
のインポートが必要
- ◆ ヒント：適宜部分項を型推論

発展2 (1/2)

- 型クラス ListLike を以下のようにさらに拡張する
 - ◆ `class ListLike f where`
 ...
 `rev :: f a -> f a`
- `nil`, `cons`, `app`, `rev`が $O(1)$ となるインスタンスを与えよ

発展2 (2/2)

- 作成したインスタンスを利用し
以下の関数と等価な処理を
 $O(n)$ で実行できるようにせよ

```
shuffle [] = []  
shuffle (x:xs) = x:reverse (shuffle xs)
```