

# 関数型プログラミング 第9回 型とクラス

---

萩野 達也

hagino@sfc.keio.ac.jp

Slide URL

<https://vu5.sfc.keio.ac.jp/slides/>

# 静的型検査と型推論

- 型

- 値の集合

- Bool = { True, False }
    - Char = { 'a', 'b', ... }
    - Int = { ... -2, -1, 0, 1, 2, 3, ... }

- 静的型検査

- Haskellはコンパイル時に式の型を検査する
  - 静的 = コンパイル時 (v.s. 動的 = 実行時)
  - すべての式は正しい型を持っている必要がある

- 型推論

- Haskellは式の型を推論する
  - 情報が不十分な場合には型推論は失敗することもある

```
main = print f  
  
f = read "80"
```



```
main = print f  
  
f :: Int  
f = read "80"
```

# 型宣言

```
var1, var2, ..., varn :: type
```

- 変数の型を明示的に宣言する
  - 型推論を助ける
  - プログラマの意図を表現する ⇒ デバッグしやすくなる

```
defaultLines :: Int  
ul, ol, li :: String -> String
```

- 式の型宣言
  - 式の中で、式の型を宣言する

```
luckyNumber = (7 :: Int)  
unluckyNumber = (13 :: Integer)
```

# 多相的(Polyomorphic)

- 型は型変数を含むことがある
- 多相的な型(Polyomorphic type)
  - 型変数を含んだ型

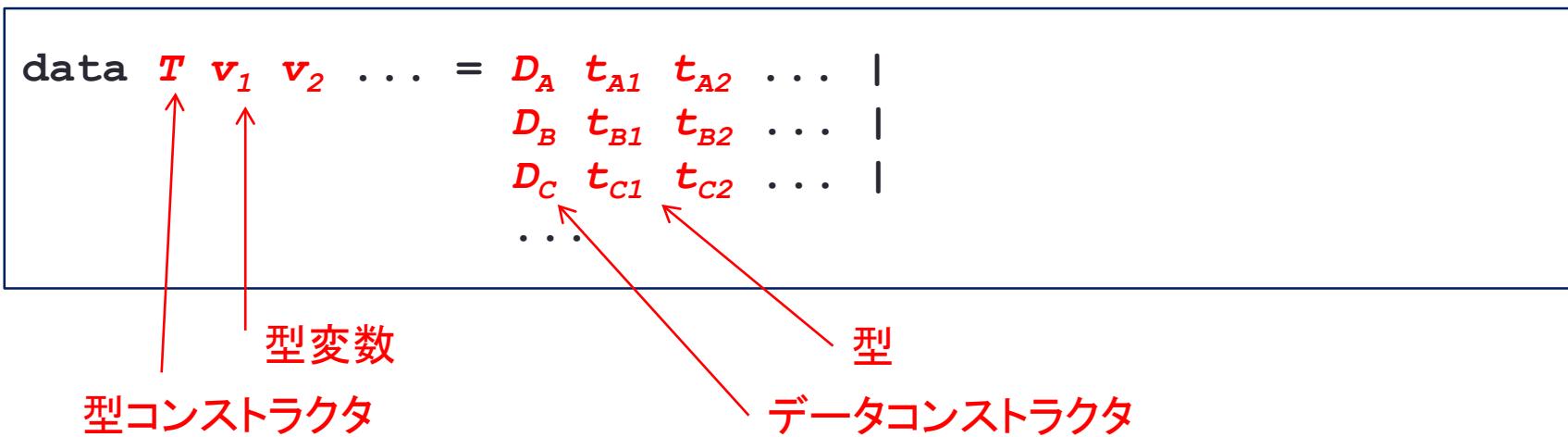
```
length :: [a] -> Int
```

```
zip :: [a] -> [b] -> [(a,b)]
```

- 型変数はどのような型にもなりうる

# 代数的データ型

- `data`宣言を使って新しい型を定義することができる.



- 型  $T$  の値はデータコンストラクタ  $D_A, D_B, D_C, \dots$  によって作られる.
- 型名, データコンストラクタは **大文字で始めなくてはいけない**.

# 例

```
data Anchor = A String String
```

- 新しい型 **Anchor** を定義
- A** が **Anchor** のデータコントラクタ
  - A** は2つの **String** フィールドを持つ
  - A** :: **String** -> **String** -> **Anchor**

```
href = A "http://www.sfc.keio.ac.jp/" "SFC Home Page"
```

- データコンストラクタパターンによってフィールドにアクセスする

```
compileAnchor (A url label) = ...
```

# フィールドラベル

- データ構造のフィールドにラベルを付けることができる

```
data Anchor = A { aURL :: String, aLabel :: String }
```

- ラベルを使ってフィールドをアクセスする

```
compileAnchor (A { aURL = u, aLabel = l }) = ...
```

```
anchorUrl (A { aURL = u }) = u
```

- フィールドラベルはセレクタとして利用できる
  - `aURL :: Anchor -> String`
  - `aLabel :: Anchor -> String`

```
href = A "http://www.sfc.keio.ac.jp/" "SFC Home Page"
```

```
main = do print (aLabel href)
```

# フィールドラベル(つづき)

- フィールドラベルを使うと、存在する値の一部のフィールドの値を変更した値を作ることができる

```
data Anchor = A { aURL :: String, aLabel :: String }

href = A "http://www.sfc.keio.ac.jp/" "SFC Home Page"

main = do { print href;
            print (href { aLabel = "that" }) }
```

「A "http://www.sfc.keio.ac.jp/" "SFC Home Page"」を出力  
「A "http://www.sfc.keio.ac.jp/" "that"」を出力

# 多相的データ型を定義

- 型変数を使うことで**多相的**データ型を定義することができる

```
data Stack a = MkStack [a]
```



型変数

```
MkStack [True, False] -- Stack Bool
```

```
MkStack ['a', 'b', 'c'] -- Stack Char
```

```
MkStack ["aa", "bb"] -- Stack String
```

# 列挙型(Enumeration Type)

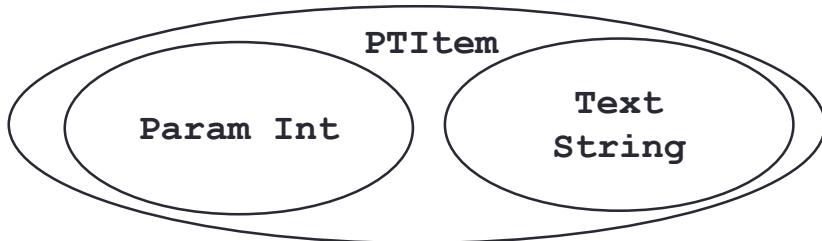
- **列挙型**を | で定義することができる

```
data OpenMode = ReadOnly | WriteOnly | ReadWrite
```

- **OpenMode** 型の値は3つのデータ構造体で作ることができる.
- **OpenMode** は3つの値を持つ:
  - **ReadOnly**
  - **WriteOnly**
  - **ReadWrite**
- **Bool** は列挙型

```
data Bool = True | False
```

# 共用体(Union)



- C言語の `union`と同じように**共用体**を定義することができる

```
data PTItem = Param Int | Text String
```

- `PTItem` の値は `Param` と整数か, `Text` と文字列のどちらかである

```

Text "daikon"
Param 5

isText::PTItem -> Bool
isText (Text _) = True
isText (Param _) = False

text::PTItem -> String
text (Text s) = s
text (Param _) = "(param)"
  
```

# 再帰的(Recursive)な型

- 型の宣言の中で自分を再帰的に使う

```
data Stack a = Empty | Push a (Stack a)
```

- Stack a の値

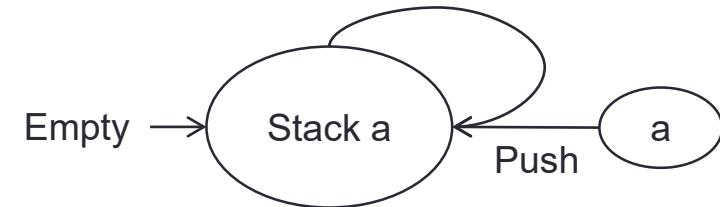
```
Empty
Push 1 Empty
Push 2 (Push 1 Empty)
Push 3 (Push 2 (Push 1 Empty))
```

- Stack a の値を参照する

```
isEmpty :: Stack a -> Bool
isEmpty Empty = True
isEmpty (Push _ _) = False
```

```
top :: Stack a -> a
top (Push x _) = x
```

```
pop :: Stack a -> Stack a
pop (Push _ s) = s
```



# type 宣言

```
type T v1 v2 ... = t
```

型コンストラクタ **T**  
 型変数 **v<sub>1</sub>** **v<sub>2</sub>** ...  
 型 **t**

- 存在する型に名前を付けることで新しい型を作る
  - データコンストラクタはなし

```
type MyList a = [a]
```

- **MyList a** は **[a]** と同じ
  - **[a]** に対する関数は **MyList a** に使うことができる

# newtype 宣言

```
newtype T v1 v2 ... = D t
```

The diagram shows the components of a newtype declaration. A horizontal line contains the text: "newtype T v<sub>1</sub> v<sub>2</sub> ... = D t". Three red arrows point from labels below the line to specific parts of the text: "type constructor" points to "T", "type variables" points to "v<sub>1</sub> v<sub>2</sub> ...", and "type data constructor" points to "D t".

- 存在する型に名前を付けることで新しい型を作る
  - データ構造体がある

```
newtype StackNT a = MKStackNT [a]
```

```
data StackNT a = MKStackNT [a]
```

- データ構造体が一つだけの `data` 宣言とほとんど同じ
  - `newtype` の方が内部表現が単純
  - `StackNT a` は単純に `[a]` として表現されている

# 型クラス

- 多相的な型の利用に制約を付ける
  - `sort :: [a] -> [a]`
  - `sort` は任意の型のリストを並び替えることができるわけではない。順序関係がないといけない。
- 型クラス(単に、クラス)
  - 型の集合
  - 型クラスの属する型はその型クラスのクラスメソッドを実装する必要がある。
- 例: `Ord` クラス
  - `Ord` クラスに属する型の値は比較することができる

```
sort :: (Ord a) => [a] -> [a]
```

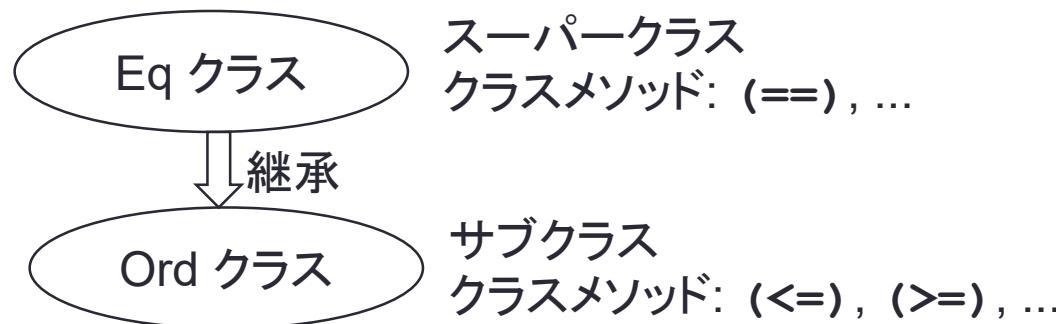
- `(Ord a) =>` は型変数 `a` に対する制約を加えている
  - `a` は `Ord` クラスに属する型でないといけない

# 継承

- クラス間には**継承関係**がある

- 例: Eq クラス

- Eq クラスは Ord クラスのスーパークラス
- Eq クラスは `(==)` がクラスメソッド



# class宣言

- Eq クラス

```
class Eq a where {
  (==), (/=) :: a -> a -> Bool    -- クラスメソッドの宣言

  x == y = not (x /= y)                -- (==) クラスメソッドのデフォルト実装
  x /= y = not (x == y)                -- (/=) クラスメソッドのデフォルト実装
}
```

- Ord クラス(Eq クラスをスーパークラスとする)

```
class (Eq a) => (Ord a) where {
  compare           :: a -> a -> Ordering
  (<), (<=), (>) , (>=) :: a -> a -> Bool
  min, max         :: a -> a -> a

  compare x y | x == y      = EQ
               | x <= y     = LT
               | otherwise   = GT
  x <= y   = compare x y /= GT
  x < y    = compare x y == LT
  x >= y   = compare x y /= LT
  x > y    = compare x y == GT
  max x y | x <= y      = y
            | otherwise   = x
  min x y | x <= y      = x
            | otherwise   = y
}
```

# instance宣言

- 型があるクラスに属していることを宣言

```
data Anchor = A String String

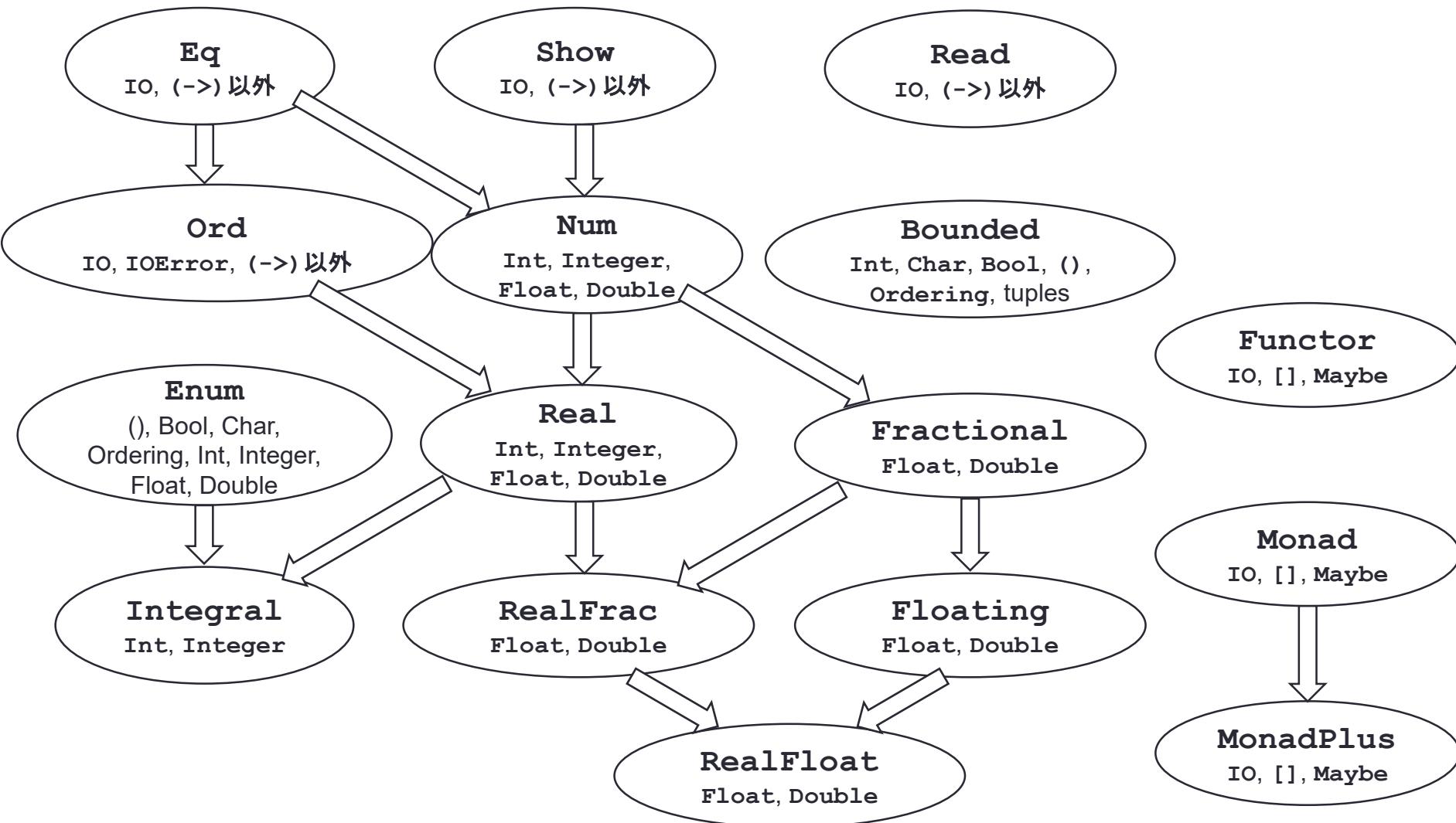
instance Eq Anchor where {
  (A u l) == (A u' l') = (u == u') && (l == l')
}
```

- **deriving** 宣言

- クラスメソッドが単純な場合にはシステムに自動実装させることもできる
- **Eq, Ord, Enum, Bounded, Show, Read**

```
data Anchor = A String String deriving (Eq, Show)
```

# いくつかのクラス



# 例：分数

- 分数は分子と分母の2つの整数からできている
  - 整数のペアとしてデータ型を宣言

```
data Rat = Rat Integer Integer

main = print $ Rat 2 3
```

- `print` することはできない
  - `print` は `show` メソッドが定義されてなくてはいけない
  - `print :: Show a => a -> IO ()`

```
data Rat = Rat Integer Integer deriving Show

main = print $ Rat 2 3
```

- これでも良いが分数の表示が「`Rat 2 3`」のままでそれらしくない
- 自分で `show` メソッドを実装する

```
data Rat = Rat Integer Integer

instance Show Rat where {
    show (Rat x y) = show x ++ "/" ++ show y
}

main = print $ Rat 2 3
```

# 分数(つづき)

- 分数の足し算や掛け算はどうすればよい？
- 四則演算の関数を定義
  - データコンストラクタパターンを利用

```
data Rat = Rat Integer Integer

instance Show Rat where {
    show (Rat x y) = show x ++ "/" ++ show y
}

add :: Rat -> Rat -> Rat
add (Rat x y) (Rat u v) = Rat (x * u + y * v) (y * v)

main = print $ add (Rat 1 2) (Rat 1 6)
```

# 分数(つづき)

- (+) や (\*) を使いたい
- Numクラスのインスタンスにする

```
class Num a where {
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a           -- あるいは (-):: a -> a -> a
  abs :: a -> a
  signum :: a -> a           -- abs x * signum x == x を満たすこと
  fromInteger :: Integer -> a
}
```

- 上記6つを実装し、RatをNumクラスのインスタンスにする

```
instance Num Rat where {
  (Rat x y) + (Rat u v) = Rat (x * u + y * v) (y * v)
  (Rat x y) * (Rat u v) = Rat (x * u) (y * v)
  negate (Rat x y) = Rat (- x) y
  abs (Rat x y) = Rat (abs x) (abs y)
  signum (Rat x y) | x == 0    = fromInteger 0
                    | x * y > 0 = fromInteger 1
                    | otherwise = fromInteger (-1)
  fromInteger x = Rat x 1
}
```

```
main = print $ Rat 1 2 + Rat 1 6
```

# 練習問題9－1

rat.hs

```
import System.Environment

data Rat = Rat Integer Integer

instance Show Rat where {
    show (Rat x y) = show x ++ "/" ++ show y
}

instance Num Rat where {
    ...
}

main = getArgs >>= \args -> test $ map read args
where test (x:y:u:v:_) =
    (print $ Rat x y + Rat u v) >>
    (print $ Rat x y - Rat u v) >>
    (print $ Rat x y * Rat u v)
```

- 次の点を修正し分数を完成させなさい
  - 分数の計算結果が既約分数になっていない
  - 分数を表示したときに分母に負の数があるのはみつともない
  - 計算結果が整数値になるのなら分数でなく整数で表示したい