関数型プログラミング

第9回　型とクラス

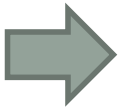
萩野　達也

hagino@sfc.keio.ac.jp

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（Polymorphic）

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

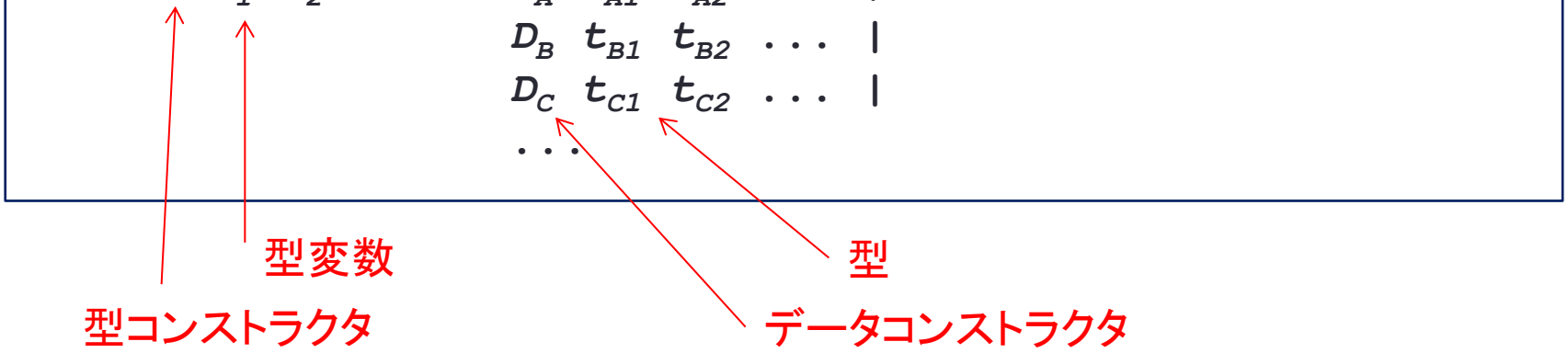
```
length :: [a] -> Int

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

- 型変数はどのような方にもなりうる

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.

```
data T v1 v2 ... = DA tA1 tA2 ... |
                   DB tB1 tB2 ... |
                   DC tC1 tC2 ... |
                   ...
```



- 型T の値はデータコンストラクタD_A, D_B, D_C,...によってるく垂れる.

- 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

- データコンストラクタ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
```

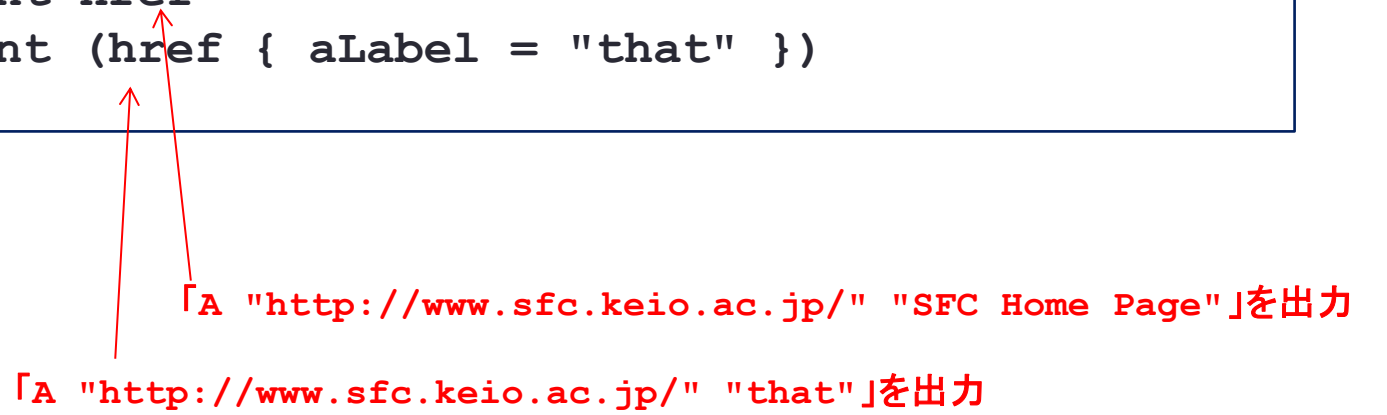
- フィールドラベルはセレクタとして利用できる
  - `aURL :: Anchor -> String`
  - `aLabel :: Anchor -> String`

```
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

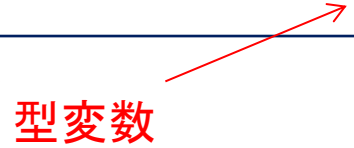
```
data Anchor = A { aURL :: String, aLabel :: String }

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

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



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]
```




```
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
```

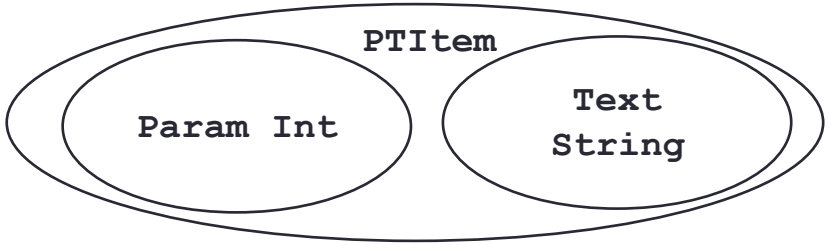
- OpenMode型の値はは3つのデータコンストラクタで作ることができる.

- **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)



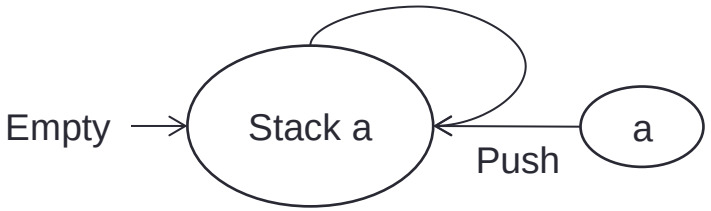
- C言語の **union** と同じように共用体を定義することができる

```
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

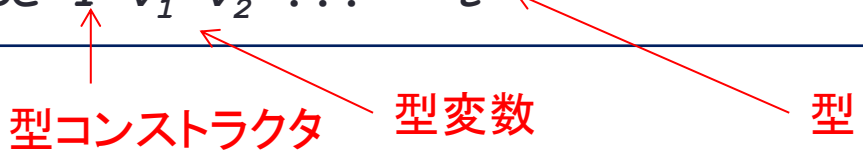
```
Emtpy
Push 1 Empty
Push 2 (Push 1 Empty)
Push 3 (Push 2 (Push 1 Empty))
```

- **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
```



- 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
```

type constructor

type variables

type
data constructor

- 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がある

```
newtype StackNT a = MKStackNT [a]

data StackNT a = MKStackNT [a]
```

- データコンストラクタが一つだけの **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

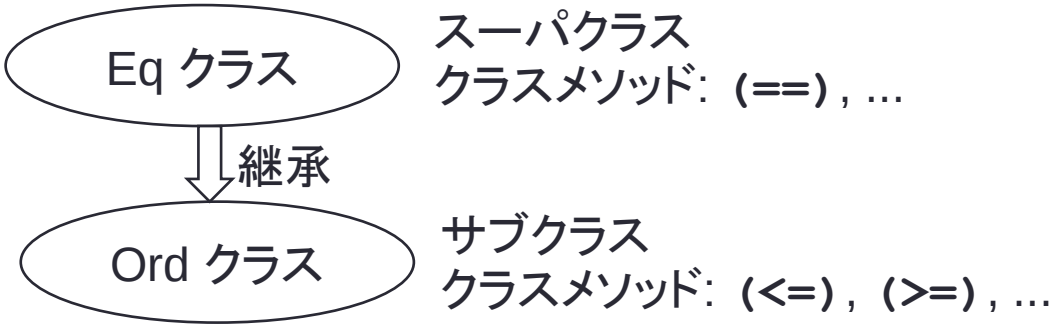
```
sort::(Ord a) => [a] -> [a]
```

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

- **Ord クラス(Eq クラスをスーパクラスとする)**

```
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

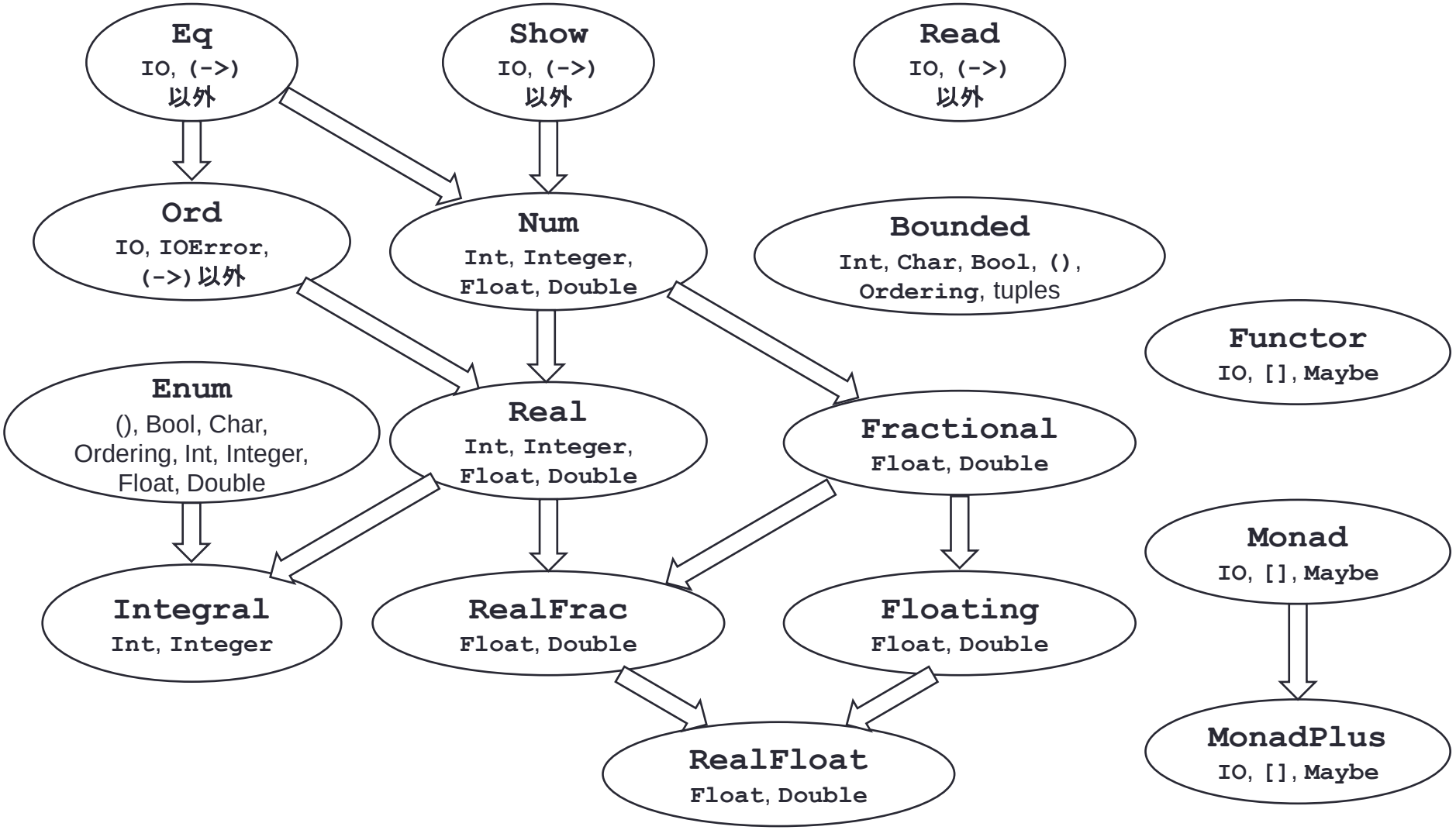
- 型があるクラスに属していることを宣言

```
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


Eq
IO, (->)
以外
Show
IO, (->)
以外
Read
IO, (->)
以外
Ord
IO, IOError,
(->) 以外
Num
Int, Integer,
Float, Double
Bounded
Int, Char, Bool, (),
Ordering, tuples
Functor
IO, [], Maybe
Enum
(), Bool, Char,
Ordering, Int, Integer,
Float, Double
Real
Int, Integer,
Float, Double
Fractional
Float, Double
Monad
IO, [], Maybe
Integral
Int, Integer
RealFrac
Float, Double
Floating
Float, Double
MonadPlus
IO, [], Maybe
RealFloat
Float, Double

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)

- フィールドを使って書き直してみる.

```
data Rat = Rat { num::Integer, den::Integer }

instance Show Rat where
  show x = show (num x) ++ "/" ++ show (den x)

main = do print $ Rat { num = 2, den = 3 }
          print $ Rat 2 3
```

- 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

```
data Rat = Rat { num::Integer, den::Integer }

instance Show Rat where
  show x = show (num x) ++ "/" ++ show (den x)

add::Rat -> Rat -> Rat
add x y = Rat { num = num x * den y + num y * den x,
                den = den x * den y }

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
  x + y = Rat { num = num x * den y + num y * den x, den = den x * den y }
  x * y = Rat { num = num x * num y, den = den x * den y }
  negate x = x { num = - num x }
  abs x = Rat { num = abs (num x), den = abs (den x) }
  signum (Rat n d) | n == 0                          = fromInteger 0
                   | n > 0 && d > 0 || n < 0 && d < 0 = fromInteger 1
                   | otherwise                       = fromInteger (-1)
  fromInteger x = Rat { num = x, den = 1 }
```

練習問題9－1

rat.hs

```
import System.Environment
import Data.Ratio

data Rat = Rat { num::Integer, den::Integer }

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

instance Num Rat where
  ...

instance Fractional Rat where
  x / y = ...
  fromRational x = Rat { num = numerator x, den = denominator x }

main = do args <- getArgs
          x <- return (read (args !! 0)::Integer)
          y <- return (read (args !! 1)::Integer)
          u <- return (read (args !! 2)::Integer)
          v <- return (read (args !! 3)::Integer)
          print $ Rat x y + Rat 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っともない
  - 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
  - 割り算はできないの？
    - **Fractional** クラスのインスタンスにする（Data.Ratioをimport）