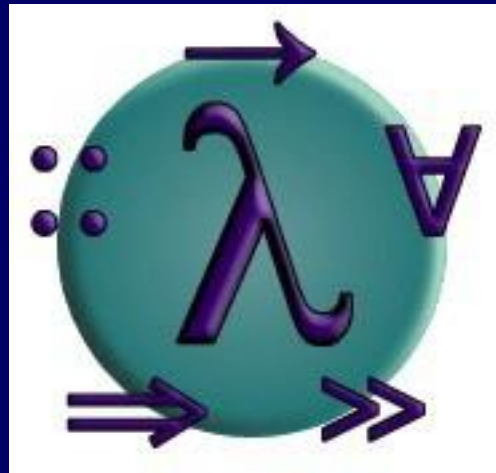


PROGRAMMING IN HASKELL



Chapter 3 - Types and Classes

愛知県立大学 情報科学部 計算機言語論(山本晋一郎・大久保弘崇、2011年)講義資料。
オリジナルは<http://www.cs.nott.ac.uk/~gmh/book.html>を参照のこと。

What is a Type?

型とは、関連する値の集まり、またそれにつける名前。
例えば、Haskellの基本型

Bool

は、以下の2つの真理値を持つ:

False

True

Type Errors

期待されている型とは異なる型の引数を関数に適用すること

```
> 1 + False  
Error
```

1は数、Falseは真理値、+は2つの数を要求する

Types in Haskell

- 式 e を評価すると型 t の値となるとき、 e は型 t を持つといい、以下のように表記する

$e :: t$

正しい式(well formed expression)は1つの型を持つ。
その型は**コンパイル時**に型推論という手続きにより自動的に決定される。

実行時ではないことに注意

- 全ての型エラーはコンパイル時に発見される。実行時に型検査をする必要がないので、より安全かつ高速にプログラムを実行できる。
- Hugsでは `:type` コマンドで式を評価せずにその型を求めることができる

```
> not False  
True
```

```
> :type not False  
not False :: Bool
```

Basic Types

Haskellは以下のような多数の基本型を持つ:

`Bool`

- logical values

`Char`

- single characters

`String`

- strings of characters

`Int`

- fixed-precision integers

`Integer`

- arbitrary-precision integers

`Float`

- floating-point numbers

リスト型(List Types)

リストは同じ型の値の並び:

```
[False, True, False] :: [Bool]
```

```
['a', 'b', 'c', 'd'] :: [Char]
```

一般に:

[t] は型 t の値を要素として持つリストの型。

[t] is the type of lists with elements of type t.

注意:

無限長のリストも許される!

- リスト型に長さの情報は含まれない:

```
[False, True] :: [Bool]
```

```
[False, True, False] :: [Bool]
```

- 要素の型に制約がない。リストのリストも作れる:

```
[[ 'a' ], [ 'b', 'c' ]] :: [[Char]]
```


タプル型(Tuple Types)

タプルは値の(有限個の)組で、各要素の型は異なっても良い:

```
(False, True)      :: (Bool, Bool)
```

```
(False, 'a', True) :: (Bool, Char, Bool)
```

一般に:

(t_1, t_2, \dots, t_n) は n 項組の型であり、
 i 番目の要素は t_i 型を持つ ($1 \leq i \leq n$)

注意:

- タプル型は長さの情報を含んでいる:

```
(False, True)      :: (Bool, Bool)
```

```
(False, True, False) :: (Bool, Bool, Bool)
```

- 要素の型に制約がない:

```
('a', (False, 'b')) :: (Char, (Bool, Char))
```

```
(True, ['a', 'b'])  :: (Bool, [Char])
```

Function Types

関数とは、ある型の値をある型の値に写像(mapping)するもの(テキストでは「写像」ではなく「変換」):

```
not      :: Bool → Bool
```

```
isDigit :: Char → Bool
```

一般に:

$t1 \rightarrow t2$ は型 $t1$ の値を型 $t2$ の値に写像する関数の型。

$t1 \rightarrow t2$ is the type of functions that map values of type $t1$ to values of type $t2$.

注意:

- 矢印 \rightarrow をキーボードで入力するときは \rightarrow とする

引数の型、戻り値の型に制限はない。例えば、引数や戻り値が複数になる関数はリストやタプルを用いる:

```
add      :: (Int,Int) → Int
add (x,y) = x+y

zeroto   :: Int → [Int]
zeroto n = [0..n]
```

カーリー化された関数(Curried Functions)

複数の引数を取る関数は、関数を返す関数を用いても表せる:

```
add'    :: Int → (Int → Int)
add' x y = x+y
```

add' は整数 x を取り、関数 $\text{add}' x$ を返す。次に、この関数は整数 y を取り、 $x+y$ の結果を返す。

注意:

add と add' は同じ最終結果を返す。ただし、add は 2 つの引数を同時に受け取り、add' は 1 つずつ受け取る:

```
add  :: (Int,Int) → Int
```

```
add' :: Int → (Int → Int)
```

Haskell Curry が果たしたこの種の関数に対する研究に敬意を表し、引数を 1 つずつ受け取る関数を **カーリー化された関数**と呼ぶ

3 つ以上の引数を取る関数も、関数を返す関数の入れ子によりカーリー化できる:

```
mult      :: Int → (Int → (Int → Int))  
mult x y z = x*y*z
```

mult は整数 x を取り、関数 $\text{mult } x$ を返す、
それは整数 y を取り、関数 $\text{mult } x y$ を返す、
それは最後に整数 z を取り、 $x*y*z$ の結果を返す

なぜカーリー化?

カーリー化された関数はタプルを取る関数よりも柔軟。
カーリー化関数に引数を**部分適用**して、有益な関数を作れる。

For example:

```
add' 1 :: Int → Int
```

```
take 5 :: [Int] → [Int]
```

```
drop 5 :: [Int] → [Int]
```


Currying Conventions

カーリー化された関数に括弧が付き過ぎるのを避けるために、2つの規則を導入:

- 型の矢印 \rightarrow は右結合(associates to the right)

$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$\text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))$ を意味する

- 矢印 \rightarrow が右結合なので、自然に関数適用は左結合(associate to the left)

```
mult x y z
```

$((\text{mult } x) y) z$ を意味する

明示的にタプルの使用が要求されない限り、Haskell の関数はカーリー化された形で定義する

多相型関数(Polymorphic Functions)

- 型変数を含む型や式を多相的という
 - polymorphic とは “of many forms” の意味
- 関数 length は多相関数、型 $[a] \rightarrow \text{Int}$ は多相型

```
length :: [a] → Int
```

任意の型 a に対して、関数 length は a 型の要素のリストを引数とし、整数を返す
注意: a がどんな型なのかまったく不明

注意:

- 型変数には、状況に応じて実際の型を当てはめる:

```
> length [False,True]  
2
```

a = Bool

```
> length [1,2,3,4]  
4
```

a = Int

- 型変数の名前は小文字で始まる。
通常 a, b, c, ...という型変数名が用いられる。

■ 標準 Prelude に含まれる多相型関数

例:

```
fst  :: (a,b) → a
```

```
head :: [a] → a
```

```
take :: Int → [a] → [a]
```

```
zip  :: [a] → [b] → [(a,b)]
```

```
id   :: a → a
```

多重定義型(値)

- 数値 3 は整数と浮動小数点の両方と加算できる
- Bool や Char とは加算できない
- 数値 3 の型は?

Num クラスのインスタンスである任意の型 t に対して、値 3 は型 t を持つ

```
Prelude> :t 3
3 :: (Num t) => t
Prelude> :t 3 + 0.0
3 + 0.0 :: (Fractional t) => t
Prelude> :t 3 + 0
3 + 0 :: (Num t) => t
Prelude> 3 + True
ERROR
```

多重定義型(関数)

- + は整数にも浮動小数点にも適用可能
- Bool や Char には適用できない
 - よって、(+) `:: a -> a -> a`ではない
- **クラス制約**: 型を限定する(ここでは数値型に限定)
- クラス制約を含む型を多重定義型という

`(+) :: Num a => a -> a -> a`

Num クラスのインスタンスである任意の型 `a` に対して、関数 `(+)` は型 `a -> a -> a` を持つ
(型の集合 Num の要素である任意の型 `a` に対して、...)

クラス: 共通のメソッドを備えた型の集合

メソッド: 多重定義された関数

多重定義関数(Overloaded Functions)

多相型関数の型が**クラス**制約を含むとき、多重定義されているという

共通のメソッドを備えた型の集合

```
sum :: Num a => [a] -> a
```

任意の数値型 a に対して、関数 `sum` は a 型の値のリストを引数とし、 a 型の値を返す

for any numeric type a , `sum` takes a list of values of type a and returns a value of type a .

注意:

制約付きの型変数には、制約を満たす型を当てはめる:

```
sum :: Num a => [a] -> a
```

```
> sum [1,2,3]  
6
```

```
> sum [1.1,2.2,3.3]  
6.6
```

```
> sum ['a','b','c']  
ERROR
```

a = Int

a = Float

Char は数値型
(Num クラスのイ
ンスタンス)では
ない

■ Haskell には多数の型クラスがある:

Num - 数値(Numeric)の型クラス

Eq - 同等性(Equality)の型クラス

Ord - 全順序(Ordered)の型クラス

■ 例:

```
(+) :: Num a => a -> a -> a
```

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

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

基本クラス(Eq - 同等クラス)

- 同等と不等を比較できる値の型の集合
 - Bool, Char, String, Int, Integer, Float などの基本型
 - 要素が Eq のインスタンスである、リストやタプルも

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

> False == False
True
> [1, 2] == [1, 2, 3]
False
> ("ab", False) /= ("ab", False)
False
```

基本クラス(Ord - 順序クラス)

- Eq クラスのインスタンスであり、かつ全順序を持つ値の型の集合
 - Bool, Char, String, Int, Integer, Float などの基本型
 - 要素が Ord のインスタンスである、リストやタプルも(辞書式)

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

```
> False < True
```

```
True
```

```
> [1, 2] < [1, 2, 3]
```

```
True
```

```
> ("ab", False) < ("ab", False)
```

```
False
```

基本クラス(ShowとRead - 表示と読込可能)

- Show: show によって文字列に変換可能
- Read: read によって文字列を値に変換可能
 - Bool, Char, String, Int, Integer, Float などの基本型
 - 要素がこのクラスのインスタンスである、リストやタプルも

```
class Show a where
    show      :: a -> String

class Read a where
    read      :: String -> a

> show 123
"123"
> read "123" :: Int
123
```

基本クラス(Num - 数値クラス)

- Eq クラスのインスタンスであり、以下の6つのメソッドによって計算可能な数値を値としてもつ型の集合
 - Int, Integer, Float などの基本型
- 注意: 除算のメソッドを備えていない

```
class (Eq a, Show a) => Num a where
  (+), (-), (*)  :: a -> a -> a
  negate        :: a -> a
  abs, signum   :: a -> a
```

```
> negate 3.3
```

```
-3.3
```

```
> signum (-3)
```

```
-1
```

基本クラス(Integral と Fractional)

- Integral: Num クラスのインスタンス、かつ整数の商と余りを計算するメソッド(div, mod)を備える
- Fractional: Num クラスのインスタンス、かつ分数の除算と逆数を計算するメソッド(/, recip)を備える

```
> 7 `div` 2
3
> 7 `mod` 2
1
> 7.0 / 2.0
3.5
> recip 2.0
0.5
```

Hints and Tips

- Haskell で関数を新しく定義するときは、最初にその型を書いてみると良い
- プログラムを書いているときに、スクリプト中の全ての関数に関して型を書くのは良い習慣である
- 多相型の関数を書くとき、その中で数値や同等性や順序を使っているなら、注意深く、必要なクラス制約を用いること

まとめ

- 型: 同じ性質を持つ値の集合
- コンパイル時の型推論によって型エラーを検出
- 基本型: Bool, Char, String, Int, Integer, Float
- リスト型: 同じ型の値の並び [t]
- タプル型: n 項組 (t1, t2, ..., tn)
- カリー化された関数:
 - $\text{add}' :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
 - 型の矢印 \rightarrow は右結合、関数適用は左結合
- 多相型: 型変数を含む型
 - $\text{length} :: [\text{a}] \rightarrow \text{Int}$
- 多重定義型: クラス制約を持つ型
 - $\text{sum} :: \text{Num a} \Rightarrow [\text{a}] \rightarrow \text{a}$