データ型

プログラム言語論

櫻井彰人

目次

- 静的型と動的型
- 型の完全性
- Haskell における型
- 単態と多態
- Hindley-Milner 型推論
- オーバーロード

https://ja.pngtree.com/freepng/

目次

- **静的型と動的型**
- 型の完全性
- Haskell における型
- 単態と多態
- Hindley-Milner 型推論
- オーバーロード

型とは何か

型エラー:

```
? 5 + [ ]
ERROR: Type error in application
*** expression : 5 + [ ]
*** term : 5
*** type : Int
*** does not match : [a]
```

型は値の集合か?

- ✓ `int = { ... -2, -1, 0, 1, 2, 3, ... }`
- ✓ `bool = { True, False }`
- ✓ `Point = { [x=0,y=0], [x=1,y=0], [x=0,y=1] ... }`

型とは何か

型は、動作(動き、作用の一部)の部分的な仕様か?

- ✓ `n,m:int ⇒ n+m` は正しいが, `not(n)` はエラー
- ✓ `n:int ⇒ n := 1` は正しいが, `n := "hello world"` はエラー

有用なまた興味深い仕様とはどのようなものであろうか?

静的な型、動的な型

値 は、プログラム言語が定義する、様々な 静的な型 を持つ.
一方、変数は、静的な型を持つと宣言されることがある.
変数と式 は、実行時にそれが持つ値で定まる 動的な型 を持つ.

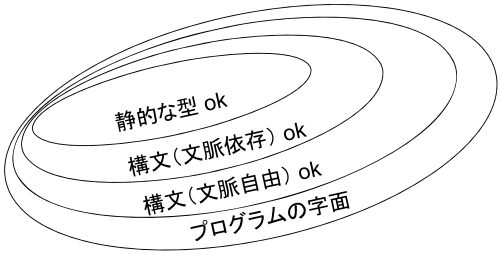
宣言した, 静的な型は `Applet`

値の静的な型は `GameApplet`

```
Applet myApplet = new GameApplet();
```

実際の動的な型は `GameApplet`

静的な型を使って、書くプログラムに制約を課す



バグのあるプログラムを排除する、
そもそも書かせない

静的な型付けと動的な型付け

ある言語が静的に型付けされているとは、任意の式の(静的な)型が、プログラムの字面だけで、常に決定できることを言う.

ある言語が 動的に片付け されているとは 値だけが固定した型を持つことを言う. 変数とパラメータ(仮引数)が実行時にとる型は、異なっても(変わっても)よい. 使われる前に、直ちにチェックする必要がある.

ある言語が "強い型付け" であるとは、オブジェクトに対する不適切な演算(操作)が行えないことをいう.

型の一貫性を担保するのは

I. コンパイル時の型チェック,
II. 型推定(type inference), または
III. 動的型チェック.

強い, 弱い, 静的, 動的

	静的	動的
強い	Java, Pascal	Smalltalk, Ruby
弱い	C	Assembler

型の種類

どのプログラム言語も、組込み型を用意している.

- ✓ **基本型(Primitive types):** booleans, integers, floats, chars ...
- ✓ **複合型(Composite types):** functions, lists, tuples ...

現代の強い型付け言語では、多くの場合、ユーザ定義型が追加されている.

- ✓ **ユーザ定義型(User-defined types):** enumerations, recursive types, generic types, objects ...

目次

型の完全性

型の完全性原理:

値の型に関しては、それに対するどのような演算(操作)も、恣意的に制限されてはいけない.

— Watt

第一級の値 は 評価され, 引数として渡され、複合型の要素として用いられる.

関数型言語ではクラス間の差異をなくそうと努力するが、命令型言語では、関数は(もっともよくて)第二級の値として扱うのが典型的である.

目次

- 静的型と動的型
- 型の完全性
- **Haskell における型**
- 単態と多態
- Hindley-Milner 型推論
- オーバーロード

関数型

関数型を用いると、式の型を、評価することなく、演繹することができる:

```
fact :: Int -> Int
42 :: Int              ⇒      fact 42 :: Int
```

カリー型:

```
Int -> Int -> Int      ≡    Int -> (Int -> Int)
```

そして

```
plus 5 6               ≡    ((plus 5) 6)
```

従って:

```
plus::Int->Int->Int    ⇒    plus 5::Int->Int
```


リスト型

型 a の値のリストが持つ型 `[a]`:

```
[ 1 ] :: [ Int ]   -- Int のリスト型
```

NB: あるリストの要素はすべて同じ型!

```
['a', 2, False]   -- 合文法的ではない。型付けできない
```


タプル型

式 `x1, x2, ..., xn` の型が, それぞれ, `t1, t2, ..., tn` である時, タプル `(x1, x2, ..., xn)` の型は `(t1, t2, ..., tn)` である:

```
(1, [2], 3) :: (Int, [Int], Int)
('a', False) :: (Char, Bool)
((1,2),(3,4)) :: ((Int, Int), (Int, Int))
```

ユニット型は `()` と書かれ、同様に `()` と書かれるユニット型を有する.

ユーザデータ型

以下 Goferに準拠

新しいデータ型は、以下のものを指定することにより導入できる

I. データ型名,
II. パラメータ型集合, そして
III. その型の要素に対する コンストラクタ

```
data DatatypeName a1 ... an = constr1 | ... | constrm
```

と書かれ、コンストラクタは次のいずれか:

1. コンストラクタ関数: `Name type1 ... typek`
2. 中置型(2進)コンストラクタ (i.e., ":" で始まるもの): `type1 BINOP type2`

例: 列挙型

データを保持しないユーザデータ型で列挙がモデル化できる:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

命名規則: 型名とコンストラクタ名は大文字で始める必要がある

ユーザデータ型上で定義される関数は、引数を分解し、コンストラクター個につき1ケースとなる場合分けをしなければならない:

```
whatShallIDo Sun  = "relax"
whatShallIDo Sat  = "go dating"
whatShallIDo _    = "guess I'll go back to school"
```

例: 直和(Union)型

```
data Temp = Centigrade Float | Fahrenheit Float

freezing :: Temp -> Bool
freezing (Centigrade temp)  = temp <= 0.0
freezing (Fahrenheit temp)  = temp <= 32.0
```


例: 再帰(Recursive)型

再帰型により、型自体の上に定義するコンストラクタが作れる:

```
data Tree a = Lf a | Tree a :^: Tree a
mytree = (Lf 12 :^: (Lf 23 :^: Lf 13)) :^: Lf 10
```

```
? :t mytree
> mytree :: Tree Int
```

mytree =

再帰型の使用例

```
leaves, leaves' :: Tree a -> [a]
leaves (Lf l)   = [l]
leaves (l :^: r)  = leaves l ++ leaves r

leaves' t = leavesAcc t [ ]
   where leavesAcc (Lf l) = (l:)
      leavesAcc (l :^: r) = leavesAcc l . leavesAcc r
```

NB: `(f . g) x = f (g x)`

- これらの関数の働きは?
- より効率的にできるのは?

目次

- 静的型と動的型
- 型の完全性
- Haskell における型
- **単態と多態**
- Hindley-Milner 型推論
- オーバーロード

単相性(単態性)

Pascal や C のような言語の型システムは 単相性: すべての定数, 変数, 引数, 関数値は ただ一つの型を持つ.

- 型チェックにはよい
- 型に依存しないプログラムを書くにはよくない
  - 例: Pascal では、型に依存しない整列プログラムは書けない
    - integer型配列用のプログラム, real型の配列用のプログラム,,,
    - 厳格に適用すると、長さが異なる毎に整列プログラムが必要

多相性(多態性)

多相関数 の引数の型は異なってもよい:

```
length          :: [a] -> Int
length [ ]      = 0
length (x:xs)   = 1 + length xs

map             :: (a -> b) -> [a] -> [b]
map f [ ]       = [ ]
map f (x:xs)    = f x : map f xs

(.)             :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x       = f (g x)
```

目次

- 静的型と動的型
- 型の完全性
- Haskell における型
- 単態と多態
- **Hindley-Milner 型推論**
- オーバーロード

型推論

式の型は、多くの場合、単にその式の構造を見るだけで 推論 することができる. 例えば:

```
length [ ]      = 0
length (x:xs)   = 1 + length xs
```

明らかに:

```
length :: a -> b
```

さらに, b は明らかに int であり, a は list である, 従って:

```
length :: [c] -> int
```

しかし、型の詳細化をこれ以上進めることはできない.

多相型の合成

式の型の導出は、多相型を用いて、単に型変数を具体的な型に結びつけるだけでできる.

例えば:

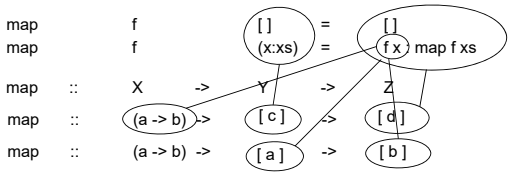
```
length   :: [a] -> Int
map   :: (a -> b) -> [a] -> [b]
```

とすると:

```
map length                    :: [[a]] -> [Int]
[ "Hello", "World" ]          :: [[Char]]
map length [ "Hello", "World" ]   :: [Int]
```


多相型推論

Hindley-Milner 型推論では、自動的に、多くの多層型の関数の型が型推論できる.



こうした型システムは、MLやHaskellを含む、現代の関数型言語で用いられている.

型の特殊化

多相型関数に、より特殊な型を割り付けることもできる:

```
idInt :: Int -> Int
idInt x = x
```

注: コマンド :t を用いると、ある特定の式の、Haskellが推定する型を表示することができる:

```
? :t \x -> [x]
> \x -> [x] :: a -> [a]

? :t (\x -> [x]) :: Char -> String
> \x -> [x] :: Char -> String
```


多相の種類

- ユニバーサルな多相性:
  - パラメトリック: Haskellでは多相 map 関数; Pascal/C における nil/void ポインタ型
  - 包含的(inclusion): 部分型(subtyping) — 使用できる、型の範囲を制限する グラフィックオブジェクト等
- アドホックな多相性:
  - オーバーロード: 例えば + は整数にも実数にも適用できる
  - コアージョン: 実数が使用できるところなら、整数も使用できるi、そして vise versa.

目次

- 静的型と動的型
- 型の完全性
- Haskell における型
- 単態と多態
- Hindley-Milner 型推論
- **オーバーロード**

コアージョン vs オーバーロード

- "+" のようにオーバーロードされる演算子がある
  - + にはいくつかの可能な型がある.
    例えば: `int +(int,int)`, `float +(float, float)`
    さらに `float* +(float*, int)`, `int* +(int, int*)`
  - それぞれ, 演算子 "+" は異なる実装となる
  - どの実装を用いるかは、演算項の型による
- 型強制(coercion): `int`と`float` に "+" が適用される.
  - 可能なすべての型に "+" を定義するのではなく, 引数を自動的に型変換する
  - 型強制にはプログラムが必要となることがある (e.g. `int` から float への型変換)
  - 型変換は、通常、より一般的な型に変換する i.e. 5+0.5 は `float` 型に (というのも `float` ≥ `int`)

コアージョン vs オーバーロード

コアージョン または オーバーロード — 区別できますか?

```
3   + 4
3.0 + 4
3   + 4.0
3.0 + 4.0
```

➢ いくつかのオーバーロードした "+" 関数があるのか, それともーつだけあって, 値が自動的に型変換されるのか?

補足

リストと配列

- プログラム字面上、違いが少ないことが多い
- しかし、型は全く異なる
  - アクセス方法が異なる
  - 要素に対する制約が異なる

深く関連している

- 実装方法も異なる
  - 速度が異なる
  - 占有メモリが異なる
  - 実行系の処理が異なる

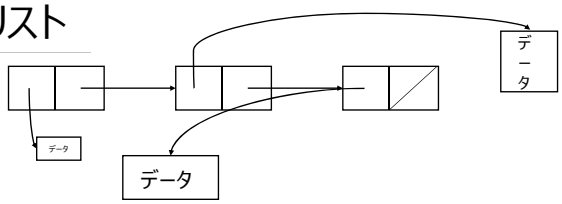
補足

配列（array）

- 添え字（index）でアクセス
- 添え字はある範囲の整数
  - 言語によって、下限は、0であったり、1であったり、
  - さらには、負の添え字が許されたりする
  - つまり、任意の整数範囲が可能なこともある
- メモリ上、連続領域を占める
- 各データの大きさが同じ
- アクセスは高速
  - ハードウェア（CPUやメモリ）のサポートあり
- データの挿入や削除は、コスト高

補足

リスト



- ポインタ（アドレス）を用いて繋いだ（ように見える）構造
- データの型は、一般に、いろいろあってよい
- メモリ上、散らばってよい
- 「先頭」と「次」を用いてアクセス
- アクセスは、（後ろの方ほど）遅い
  - 逆方向の「リンク」があれば、真ん中が遅い
- 追加、削除は速い