関数型（function type）を見つめるプログラミング

山下 伸夫 nobsun@sampou.org

2019-11-09

関数型を見つめる

看板に偽りあり: 「関数型（function type）を見つめる ~~プログラミング~~」

> 拡張適用演算子を考え続けている

ゆるふわ（あたり前のことを言い換えただけ？）の話

- 2項演算子は高階関数だよね．余域が関数型の高階関数は2項演算子だよね．
- `curry` は高階化関数だよね． `` `curry` `` は部分適用演算子だよね．
- `map` は関数拡張関数だよね． `` `map` `` は拡張適用演算子だよね．
- `map` あるんなら `ap :: [a -> b] -> ([a] -> [b])` も欲しくなるよね．
- `` `ap` `` は非関数を引数に適用する拡張適用演算子だよね．
- `($)` は何もしてないから，`id` でいいよね．
- 2項演算子は拡張適用演算子ということでいいよね．

リスト型（list type）

> τ が型なら，$[\tau]$ は型

組型（tuple type）

> σ および τ が型なら，(σ, τ) は型

関数型（function type）

域 (domain) が σ，余域 (codomain) が τ であるような関数の型

> σ および τ が型なら，$\sigma \to \tau$ は型

高階関数型

域が関数型であるような高階関数の型

> σ_1，σ_2 が型なら，$\sigma_1 \to \sigma_2$ は型．
>
> だから
>
> τ が型ならば，$(\sigma_1 \to \sigma_2) \to \tau$ は型

余域が関数型であるような高階関数の型

> τ_1，τ_2 が型なら，$\tau_1 \to \tau_2$ は型
>
> だから
>
> σ が型ならば，$\sigma \to (\tau_1 \to \tau_2)$ は型

2変数関数型

> σ および τ が型なら，(σ, τ) は型
> だから
> υ が型なら，$(\sigma, \tau) \to \upsilon$ は型

2項演算子（のセクション）は高階関数

余域が関数型になるような高階関数

> 関数を構成する関数

あまり意識されないように思えるがとても**大切**

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

`(+)` は関数で，域は `a` ，余域は `a -> a`

余域が関数型の高階関数は2項演算子

```
f :: a -> b -> c

(x :: a) `f` (y :: b) :: c
```

`curry` は高階化関数

```
>>> :t curry
curry :: ((a, b) -> c) -> a -> b -> c
```

関数 `curry` において，域は `(a, b) -> c` で2変数関数型，余域は `a -> (b -> c)` で高階関数型

`` `curry` `` は部分適用演算子

```
f `curry` x :: b -> c
```

`` `curry` `` の左オペランドは `(a, b) -> c` の2変数関数型の値，右オペランドは `a` 型の値．演算結果は，残りの `b` 型の引数を待つ `b -> c` 型の関数．

演算子 `` `curry` `` は `f` を `x :: a` に部分適用する．

`map` は関数拡張関数

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

関数 `map` において，域は `a -> b` で関数型，余域は `[a] -> [b]` で域の関数型を拡張した関数型．

`fmap` は一般化した `map`

```
fmap :: Functor f => (a -> b) -> (f a -> f b)
```

`` `map` `` は拡張適用演算子

```
f `map` xs :: [b]
```

`map` の左オペランドは `a -> b` の関数型の値，右オペランドは `a` を拡張したリスト `[a]` 型の値．演算喧嘩は， `b` を拡張したリスト `[b]` 型の値．

演算子 `` `map` `` は `f :: a -> b` を `[a]` に拡張適用する．

`<$>` は一般化した `` `map` ``

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
```

`ap :: [b -> c] -> ([b] -> [c])` も欲しい

```
(f :: a -> (b -> c)) `map` (xs :: [a]) :: [b -> c]
```

`f :: a -> (b -> c)` のように余域が関数型であるような高階関数を `xs :: [a]` に `` `map` `` で拡張適用すると `[b -> c]` という関数のリスト型の値になる．これを `[b] -> [c]` という関数に変換したい．

```
ap :: [a -> b] -> ([a] -> [b])
ap fs = \ xs -> [f x | f <- fs, x <- xs]
```

`ap` は非関数を引数に適用する拡張適用演算子

```
(fs :: [a -> b]) `ap` (xs :: [a]) :: [b]
{- ^ 非関数   -}
```

`<*>` は一般化した `ap`

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

`($)` は何もしてないから `id`

`$` は関数適用演算子で，

```
f $ x = f x
```

`($)` は域が `a -> b` ，余域が `a -> b` の関数変換関数．

```
($) :: (a -> b) -> a -> b
($) f = f
```

なら

```
($) = id
```

2項演算子は拡張適用演算子

任意の2項演算子 ⊙ すなわち `(⊙) :: a -> b -> c` について

```
(⊙) = ($) . (⊙) = (⊙) . ($)
```

だから，

> 2項演算子は拡張適用演算子