


# プログラミング言語論 #13

## Partial Application

2018-07-17





関数の arity に関する制約は本当に必要だろうか、緩和できないだろうか。

# 準備：計算の順序

## 演算子の結合性

$$2 - 4 - 5 = 2 - (4 - 5) \text{ or } (2 - 4) - 5$$

$$10 \wedge 3 \wedge 2 = (10^3)^2 \text{ or } 10^{(3^2)}$$

同じ演算子を複数回含む式には正しい計算の順序がある  
これを演算子の**結合性** [▶ wikipedia](#) という

演算子ごとに右結合か左結合かが決まっている

## 高階関数の復習

前回 (->) に対して括弧を付けることで、関数が関数の引数・返値になりうることを見た.

`f :: Int -> Int -> Int`

`g :: (Int -> Int) -> Int`

`h :: Int -> (Int -> Int)`

前回導入したのは、括弧が付いている型をどのように解釈すべきかである. 括弧を後から付けられるかどうかは検討していない.

※ どう考えても構文的に (->) は型に関する演算子である. 結合性があるはず.

## 演算子 (->) に対する結合性の検討

問題：下式に対して、結合性を考慮した括弧の後付けは可能か.

```
1 Int -> Int -> Int
```

別の表現：演算子 (->) は左結合か、右結合か

解釈 1 できる, 左結合:  $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$  ならば  $(\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$

解釈 2 できる, 右結合:  $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$  ならば  $\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$

解釈 3 できない:  $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$  は他の形に変形できない

どれが妥当か. 妥当なものが複数あるなら各言語でどのルールを使っているか.

# 解釈 1

左の演算を優先

$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$  ならば  $(\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$

これは  $(\rightarrow)$  は左結合であることを含意する.

- 左辺は「2 つの `Int` 型の引数を取る関数」を意味する
- 右辺は「1 つの関数を引数として取る関数」を意味する

ある関数が左辺の型を持つならば自動的に右辺の型を同時に持つと解釈できなければならない.

```
1 f 3 5      -- エラーが起きない
2 f negate -- エラーが起きない (cf negate :: Num a => a -> a)
```

## 解釈 2

右の演算を優先

$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$  ならば  $\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$

これは (->) は右結合であることを含意する.

- 左辺は「2 つの Int 型の引数を取り Int 型を返す関数」を意味する
- 右辺は「Int 型の引数を取り, 『Int 型の引数を取り Int 型を返す関数』を返す関数」を意味する

この解釈は左結合のものよりは受け入れやすい.

なんと言っても Int 型の第 1 引数を取るところまでは左辺も右辺も共通である.

Haskell はこの解釈を取る. すなわち, 2 引数の関数に 1 引数しか与えなくてもそれが直接の型エラーを起こすことはない.

## 解釈 2 の結論の意味するもの

```
foo :: Int -> Int -> Int
```

```
1 x = foo 2 3    -- エラーでない
```

```
foo :: 上と等価
```

```
1 y = foo 3      -- エラーでない
```

どちらも型エラーにならない



## 解釈3：括弧を追加することは出来ない

$\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$  は括弧を付けた別の形には等価変換できない

これは2引数関数と1引数関数はまったく別物で、変換の余地はないことを意味する。

- 2引数関数に3つの実引数を与えることは型エラー
- 2引数関数に1つのみの実引数を与えることは型エラー

Cを始めとする多くの言語はこの立場を取る。

## Haskell の場合の再確認

演算子  $(->)$  は右結合であるため、括弧は右寄せで自由に付けることができる。

```
1 f :: Int -> Int -> Int -> Int
2   = Int -> Int -> (Int -> Int)
3   = Int -> (Int -> (Int -> Int))
4 f x y z = (100 * x) + (10 * y) + z
```

```
1 :type f
2 :type (f 1)
3 :type (f 1 2)
4 :type (f 1 2 3)
```

すなわち  $\text{arity } N$  の関数は  $k$  個の実引数を与えると  $\text{arity } N - k$  の関数を返す高階関数になる。特別な場合として  $N$  個の実引数を与えた場合に  $\text{arity } 0$ ，すなわち関数ではなくただの値になる。

# 引数の数が少なくてもよいと考える理由

## 仮引数と実引数の対消滅

f 0 1 2 とは関数 f に Int, Int, Int の引数を与えること：

```
1 Int -> Int -> Int -> Int <- Int <- Int <- Int
```

この時、以下のことが起きていると考えられる：

$$\begin{aligned} & Int \rightarrow Int \rightarrow Int \rightarrow Int \leftarrow Int \leftarrow Int \leftarrow Int \\ &= Int \rightarrow (Int \rightarrow Int \rightarrow Int) \leftarrow Int \leftarrow Int \leftarrow Int \\ &= (Int \rightarrow Int \rightarrow Int) \leftarrow Int \leftarrow Int \\ &= Int \rightarrow (Int \rightarrow Int) \leftarrow Int \leftarrow Int \\ &= Int \rightarrow Int \leftarrow Int \\ &= Int \end{aligned}$$

1 行目から 2 行目, 3 行目から 4 行目への変形において (->) の右結合性を利用した.

右向き矢印の数と左向き矢印の数が一致すると一つの値になる (対消滅)

## 引数の数が少なくてもよいと考える理由

ならば、 $f\ 0\ 1$  のように関数  $f$  に  $\text{Int}$ ,  $\text{Int}$  の引数を与えると：

$$\begin{aligned} & \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \leftarrow \text{Int} \leftarrow \text{Int} \\ &= \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}) \leftarrow \text{Int} \leftarrow \text{Int} \\ &= (\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}) \leftarrow \text{Int} \\ &= \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \leftarrow \text{Int} \\ &= \text{Int} \rightarrow \text{Int} \end{aligned}$$

これ以上変形できない．結果は 1 引数関数になった．

## 補足

Haskell においては以下の等式が成立する

$$a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)$$

ので、以下（右辺から左辺への変換）も成立する：

系： Haskell （の型記述）においては最右括弧は外せる

## プログラミング言語論専門用語 16

### partial application, 部分適用

関数の arity よりも少ない引数を与えた関数適用により，残りの arity を持つ関数を作り出し，それを返すこと

既に存在していた関数の定義を変えるのではなく，部分適用した瞬間に新しい関数がメモリ上に作られると考えよ．専門的には関数閉包という．

高階関数はアルゴリズムの分割記述を提供することでライブラリ関数の汎用化を目指す．部分適用は汎用的な関数から特定化された関数を作り出す逆向きの作用を持つ．

## C の場合

- arity の違いを許さない
- 関数は全てプログラマが定義するものであり、実行中に生成されるものではない

部分適用はできてない。 8 行目が面白いだけ。

```
1  #include <stdio.h>
2  int div(int, int);          // 2引数関数
3  int mod(int, int);          // 2引数関数
4  int (*s(int))(int, int);    // 2引数関数を返す1引数関数
5
6  int main ()
7  {
8      int x = s(0)(33, 10);    // f = s(0); x = f(33, 10);
9      return printf("%d\n", x);
10 }
11
12 int div (int x, int y) { return x / y; }
13 int mod (int x, int y) { return x % y; }
14 int (*s(int x))(int, int) { return x == 1 ? div : mod; }
```

## more examples

これは何文なのか

```
1 f = div 100      -- 変数定義文？ 代入文？ 関数定義？
```

```
1 filter :: (a -> Bool) -> [a] -> [a]
2 filter even [1, 3, 5, 8, 2, 0]      -- even :: Int -> Bool
3 filter even [9, 23, 101, 3]      -- 正確にはNum a => a->Bool
4 filter even [-1, 3, 5, -8, 2, 0]
```

これは何文なのか

```
1 e = filter even  -- eは変数か関数か
```

見慣れた関数定義文のかたち

```
1 e l = filter even l
```

左辺を見れば 1 引数関数の定義だとすぐわかる。一方、部分適用だと代入文や変数定義文にしか見えない。