プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 4

目次

ラムダ記法

関数を指定する時、その入力 (引数) を明示する記法:

- 「$f(x) = ax^2 + bx + c$ となる関数 f」
- 「x に対して、$ax^2 + bx + c$ を対応付ける関数」

関数を表す記法: $\lambda x.\ M$

- λ は、ギリシャ文字のラムダ (lambda)
- $x \mapsto M$ という対応付け
- $\lambda x.\ x^2 + 1$ は、「$f(x) = x^2 + 1$ となる関数 f」のこと。
- $f(3) = (\lambda x.\ x^2 + 1)\ (3) = 3^2 + 1 = 10$

表記を増やしただけか？

- Yes.
- But, 「関数」をデータとして表せるようになった.

ラムダ計算における「計算」

計算=簡約 (reduction): $(\lambda x.M)\ N \hookrightarrow M\{x := N\}$

例 1: $M = \lambda x.\ x + 2$ のとき ($f(x) = x + 2$ となる f のこと)、

- $f(f(3)) \hookrightarrow f(3+2) \hookrightarrow (3+2)+2$
- $M\ (M\ 3) = M\ ((\lambda x.\ x+2)\ 3) \hookrightarrow M\ (3+2) =$
  $(\lambda x.\ x+2)\ (3+2) \hookrightarrow (3+2)+2$

例 2: $N = \lambda x.\ x\ (x\ 3)$ のとき ($g(x) = x(x(3))$ となる g のこと)、

- $g(f) \hookrightarrow f(f(3)) \hookrightarrow \cdots \hookrightarrow (3+2)+2$
- $N\ M = (\lambda x.\ x\ (x\ 3))\ M \hookrightarrow M\ (M\ 3) \hookrightarrow \cdots \hookrightarrow (3+2)+2$

例 2 の N は、引数が関数である ⇒高階関数 (higher-order function)

高階関数

例 1. 関数の繰返し適用: $D = \lambda f.\ (\lambda y.\ (f\ (f\ y)))$

- $M = \lambda x.\ x + 2$ のとき
- $(D\ M)\ 3 = ((\lambda f.\ \lambda y.\ f\ (f\ y))\ M)\ 3 \hookrightarrow (\lambda x.\ M\ (M\ y))\ 3 \hookrightarrow M\ (M\ 3) \hookrightarrow (3 + 2) + 2$

例 2. 関数の合成: $C = \lambda g.(\lambda f.(\lambda y.(g\ (f\ y))))$

- $M = \lambda x.\ x + 2$ のとき ($f(x) = x + 2$ となる f のこと)
- $N = \lambda x.\ x * 3$ のとき ($g(x) = x * 3$ となる g のこと)
- 合成関数 $g \circ f$ は、$g \circ f\ (y) = (y + 2) * 3$ となる関数)
- $(C\ N)\ M \hookrightarrow \lambda y.\ N\ (M\ y) \hookrightarrow \lambda y.\ N\ (y + 2) \hookrightarrow \lambda y.\ (y + 2) * 3$

ラムダ計算と OCaml

関数型言語はラムダ計算に基づいて設計されている。

- $\lambda x.M$ を OCaml では `fun x -> M` と書く。
- $\lambda x.M$ を Scheme では `(lambda (x) M)` と書く。

OCaml での高階関数: 前述の $D = \lambda f.\ (\lambda x.\ (f\ (f\ x)))$

```
let d = fun f → fun x → f (f x);;
let m = fun x → x + 2;;
d m 3;;   (* = 7 *)
```

OCaml での等価な記法:

```
let d f x = f (f x);;
let m x = x + 2;;
d m 3;;   (* = 7 *)
```

ラムダ計算の表現力

事実: 非常に強力 (Turing 完全)

- 適切なエンコーディングにより、「再帰関数」や「自然数」や「自然数上の計算」などを全て表現することができる。
- 自然数上の (部分) 関数のうち Turing 機械で表現できるものは全て表現できる。(逆も真)
- どんなプログラミング言語でも原理的には表現できる。

ラムダ計算のまとめ

純粋な (一番基本的な) ラムダ計算の体系 [Church 1930s]

- 「関数」の概念 (のみ) を表現できるようにした計算体系
- 関数型言語の理論的基盤となっている
- Turing 完全である
- (今日の授業まででは) 「始域」や「終域」を考えていない。
  - 関数 $f : \mathbb{N} \to \mathbb{R}$ に対して、
  - 始域 (domain) は $\mathbb{N}$
  - 終域 (codomain) は $\mathbb{R}$
  - $\lambda x.\ (x\ x)$ のような式が書けてしまう。

$\Rightarrow$ 来週は、型付きラムダ計算を勉強する。

第２週演習問題 (後半)

map,fold などは高階関数の例である。

```
List.map (fun x → x + 1) [3;4;5];;
(* = [4;5;6] *)
```

map は List モジュールのライブラリ関数として与えられているが、それと同じ働きをする関数を自分で再帰関数として定義することもできる。それをやってみなさい。
さらに、余力がある人は、「整数から整数への関数 f」と「整数のリストのリスト L」をもらって、「L の要素の要素すべてに f を適用する関数 map2 を定義せよ。
例: `map2 (fun x -> x+1) [[1;2]; [5;1;6]] = [[2;3]; [6;2;7]]`

目次

コンビネータ

コンビネータ (combinator) は、閉じたラムダ式 (自由変数を持たないラムダ式) のこと。

- $\lambda x.\lambda y.\ (x\ y)\ y$... 閉じたラムダ式
- $\lambda x.\lambda y.\ (x\ y)\ z$... 閉じていないラムダ式
- $(\lambda x.x)\ x$... 閉じていないラムダ式
- $\lambda x.\ ((\lambda x.x)\ x)$... 閉じたラムダ式

代表的なコンビネータ

代表的な (よく使う、有益な) コンビネータをいくつか紹介する。

$$I = \lambda x.x \qquad \text{恒等関数}$$
$$K = \lambda x.\lambda y.x$$
$$S = \lambda x.\lambda y.\lambda z.\ (x\ z)\ (y\ z)$$
$$B = \lambda x.\lambda y.\lambda z.\ x\ (y\ z) \qquad \text{関数合成}$$
$$C = \lambda x.\lambda y.\lambda z.\ (x\ z)\ y$$

面白い性質:

- S と K だけで、全ての閉じたラムダ式と同等な式を表現できる。
- 例: $\lambda x.\lambda y.\ x\ (x\ y)$ は、$S\ (S\ (K\ S)\ K)\ I$ と同等
- $(\lambda x.\lambda y.\ x\ (x\ y))\ M\ N \hookrightarrow M\ (M\ N)$
- $(S\ (S\ (K\ S)\ K)\ I)\ M\ N \hookrightarrow (S\ (K\ S)\ K\ M)\ (I\ M)\ N \hookrightarrow$
  $(K\ S\ M)\ (K\ M)\ (I\ M)\ N \hookrightarrow S\ (K\ M)\ (I\ M)\ N \hookrightarrow$
  $(K\ M\ N)\ (I\ M\ N) \hookrightarrow M\ (I\ M\ N) \hookrightarrow M\ (M\ N)$

代表的なコンビネータ

面白い性質 2:

- これらは全て、「型」をつけることができる (来週の話題)。
- これらは全て、「型」をつけると、命題論理の「トートロジー」(恒真な論理式) に対応する。
- 型をつけた S と K からなる世界は直観主義論理 (intuitionistic logic) に対応
- 型をつけた B と C と I からなる世界は線形論理 (linear logic) に対応