

# 計算と論理

Jacques Garrigue, 2023 年 11 月 1 日

## 4 ラムダ計算

ドイツの論理学者 Schönfinkel が 1920 年代に作った論理コンビネータは元々論理の証明論のための道具だった。1930 年代に Curry と Church がそれを  $\lambda$  計算に発展させると、論理的な意味と同時に計算的な意味を持つようになった。その後、計算機科学との関係がだんだん深くなり、1950 年代からリスプから始まる関数型言語の重要な基礎となった。簡単な定義で表現力が強いので、一般的にプログラミング言語のモデルとして使われる。

### 4.1 項書換系

元々はそうではなかったが、 $\lambda$  計算は項書換系によって定義すると分かりやすい。計算を項の一部の書き換えとして見る、ということだ。例えば、数式の数換系を次のように定義できる。

項 (term)  $E ::= \mathcal{R} \mid (E + E) \mid (E - E) \mid (E \times E) \mid (E/E)$

書換規則  $x, y$  は  $\mathcal{R}$  の実数ならば、

$$(x + y) \rightarrow x + y$$

$$(x - y) \rightarrow x - y$$

$$(x \times y) \rightarrow x \times y$$

$$(x/y) \rightarrow x/y$$

$(x + y)$  は数式だが、 $x + y$  は実数である)

上の規則を簡約規則ともいう。

#### 簡約の例

$$(15 + (1/4)) \times (5 - 2) \rightarrow (15 + 0.25) \times (5 - 2) \rightarrow (15.25) \times 3 \rightarrow 45.75$$

### 4.2 $\lambda$ 計算の構文 (syntax)

定義 4.1  $\lambda$  項 ( $\lambda$ -term) は次の三つの構文からできている。

$$\begin{aligned} M &::= x && \text{変数 (variable)} \\ &| \lambda x.M && \text{抽象 (abstraction)} \\ &| M M && \text{適用 (application)} \end{aligned}$$

$x$  は文脈 (環境) で定義されている値 (別の  $\lambda$  項) を指す。

$\lambda x.M$  は項  $M$  の中に使われている変数  $x$  を束縛する。関数として見れば  $f = \lambda x.M$  という定義は、普段の  $f(x) = M$ 。ただし、 $\lambda x.M$  と書くとき、名前をつける必要がないという便利さがある。

$M_1 M_2$  は関数の適用である。普段の  $M_1(M_2)$  にあたるが、 $M_1$  は名前 (変数) だけでなく、どの  $\lambda$  項でもよい。

上の文法を普通の数式と混ぜると、

$$f(x) = x + 1 \text{ のとき } f(2)$$

が1回で書ける。

$$(\lambda x. x + 1) 2$$

**括弧の使い方** 構文では括弧を書かなかったが、曖昧性を解決するのに適宜括弧を使う。

$$\begin{aligned} \lambda x. M N &= \lambda x. (M N) \\ \lambda x. \lambda y. M &= \lambda x. (\lambda y. M) \\ M N P &= ((M N) P) \end{aligned}$$

**自由変数**  $\lambda x. M$  では、 $M$  中の全ての  $x$  の出現は「束縛されている」という。ある項  $M$  の中で、 $x$  が使われながら、束縛されていないのであれば、 $x$  は  $M$  の自由変数である。 $M$  の自由変数の集合  $FV(M)$  は帰納的に次の3条で定義される。

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x. M) &= FV(M) \setminus \{x\} \\ FV(M N) &= FV(M) \cup FV(N) \end{aligned}$$

**代入** 代入は自由変数の値を変える。 $([N/x]M)$  は項  $M$  中に出現するすべての自由な  $x$  を  $N$  で置き換える。

$$\begin{aligned} ([N/x]x) &= N \\ ([N/x]y) &= y && y \neq x \\ ([N/x]\lambda y. M) &= \lambda y. ([N/x]M) && y \notin FV(N) \cup \{x\} \\ ([N/x](M M')) &= (([N/x]M) ([N/x]M')) \end{aligned}$$

3条目では、 $y$  は  $N$  の自由変数であってはならない。そのためには、次の $\alpha$ 変換が許される。 $z$  は  $N$  の自由変数でなければ、

$$\lambda y. N \leftrightarrow \lambda z. ([z/y]N)$$

こういう束縛変数の改称は常に許される。

### 4.3 簡約規則

**定義 4.2**  $\lambda$  項、 $\alpha$  変換と次の  $\beta$  簡約で構成される項書き換え系は  $\lambda$  計算という。

$$((\lambda x. M) N) \rightarrow ([N/x]M)$$

#### 例題 4.1 簡約の例

$$\begin{aligned} &(\lambda f. \lambda g. \lambda x. f x (g x)) (\lambda x. \lambda y. x) (\lambda x. \lambda y. x) \\ \rightarrow &\lambda x. ((\lambda x. \lambda y. x) x ((\lambda x. \lambda y. x) x)) \\ \rightarrow &\lambda x. ((\lambda y. x) (\lambda y. x)) \\ \rightarrow &\lambda x. x \end{aligned}$$

$$\begin{aligned} &(\lambda x. (x x)) (\lambda x. (x x)) \\ \rightarrow &(\lambda x. (x x)) (\lambda x. (x x)) \\ \rightarrow &\dots \end{aligned}$$

**定理 4.1**  $\lambda$  計算は合流性を持つ。  $M \rightarrow \dots \rightarrow N$  と  $M \rightarrow \dots \rightarrow P$  という二つの簡約列があれば、  $N \rightarrow \dots \rightarrow T, P \rightarrow \dots \rightarrow T$  となるような  $T$  が存在する。

#### 4.4 $\lambda$ 計算は万能である

全てのプログラムは  $\lambda$  計算で書ける。

**自然数** チャーチによるエンコーディングがある (Church numeral)。

$$\begin{aligned} c_n &= \lambda f. \lambda x. (f \dots (f x) \dots) && f \text{ を } n \text{ 回適用する} \\ c_+ &= \lambda m. \lambda n. \lambda f. \lambda x. (m f (n f x)) && \text{加算} \\ c_\times &= \lambda m. \lambda n. \lambda f. (m (n f)) && \text{積算} \end{aligned}$$

**練習問題 4.1** 1.  $(c_3 (\lambda x. x x) y)$  の標準形を計算せよ。

2.  $(c_2 c_3)$  の標準形を計算せよ。

3.  $m^n$  を計算する  $\lambda$  項  $c_{\text{pow}}$  を与えよ。

**ブール環** 次のエンコーディングを使う。

$$t = \lambda x. \lambda y. x \qquad f = \lambda x. \lambda y. y \qquad \text{not} = \lambda b. \lambda x. \lambda y. (b y x)$$

数の区別も可能で、例えば次の項が引き数に数をもらい、真偽値を返す。

$$\text{if0} = \lambda n. (n (\lambda x. f) t)$$

**組**  $\lambda$  計算では組が簡単に表現できる。

$$\text{pair} = \lambda x. \lambda y. \lambda f. (f x y) \qquad \text{fst} = \lambda p. (p t) \qquad \text{snd} = \lambda p. (p f)$$

計算して見ると、

$$\text{fst} (\text{pair } a b) \rightarrow \text{pair } a b t \rightarrow (t a b) \rightarrow a$$

**引き算** チャーチエンコーディングによる引き算は比較的難しい。ここに一つの定義を与える。

$$\begin{aligned} c_- &= \lambda m. \lambda n. (n p m) \\ s &= \lambda n. \lambda f. \lambda x. (f (n f x)) \\ s' &= \lambda x. (\text{pair} (\text{snd } x) (s (\text{snd } x))) \\ p &= \lambda n. (\text{fst} (n s' (\text{pair } c_0 c_0))) \end{aligned}$$

$s$  は足す 1 を計算し、 $p$  は引く 1 を計算する。足し算は既に定義したので、 $s$  はすぐ定義できるが、 $p$  のためには補助関数  $s'$  を使う。

$s'$  は自然数の組  $(m, n)$  に対して  $(n, m + 1)$  を返す関数である。それを  $(0, 0)$  に繰り返し適用すると、順番に  $(0, 1), (1, 2), (2, 3) \dots$  が得られる。第一要素がちょうど適用回数引く 1 という値になるので、それを返すと前者関数になる。

最後に、 $m - n$  を計算するために、 $m$  に  $n$  回  $p$  を適用すればいい。よって、 $m \geq n$  ならば、

$$c_- c_m c_n \xrightarrow{*} c_{m-n}$$

**不動点演算子** 再起的な関数を定義するのに、不動点演算子  $Y$  が必要になる。その基本的な属性は、 $(Y M)$  が  $(M (Y M))$  に簡約できることである。

$$Y = (\lambda f. \lambda x. (x (f f x))) (\lambda f. \lambda x. (x (f f x)))$$

$Y$  はループを回す回数分からないとき必要になる。例えば、階数の再帰的な定義は次の通りである。

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \quad \text{if } n > 0 \end{aligned}$$

それを  $\lambda$  計算に翻訳するとこうなる。

$$c_! = \lambda n. \text{if0 } n \ c_1 \ (c_{\times} \ n \ (c_! \ (p \ n)))$$

こういう再帰的な定義は ( $c_!$  は右にも出ている)  $\lambda$  計算では直接にできないので、 $Y$  を使う。

$$c_! = Y(\lambda f. \lambda n. \text{if0 } n \ c_1 \ (c_{\times} \ n \ (f \ (p \ n))))$$

$YM \rightarrow M(YM)$  というのを思い出せば、上の方程式が成り立つことが分かる。

$$c_! \rightarrow (\lambda f. \lambda n. \text{if0 } n \ c_1 \ (c_{\times} \ n \ (f \ (p \ n)))) \ c_! \rightarrow \lambda n. \text{if0 } n \ c_1 \ (c_{\times} \ n \ (c_! \ (p \ n)))$$

因みに、階乗関数では繰り返しの回数があらかじめ分かるので、 $Y$  を使わなくても定義できる。

$$c_! = \lambda n. \text{snd } (n \ (\lambda x. \text{pair } (s \ (\text{fst } x)) \ (c_{\times} \ (\text{fst } x) \ (\text{snd } x)))) \ (\text{pair } c_1 \ c_1))$$

しかし、不動点を使えば、止まるかどうか分からない関数も定義できる。

$$Y \ (\lambda f. \lambda x. s \ (f \ x)) \ c_1 \rightarrow s \ (Y \ (\lambda f. \lambda x. s \ (f \ x)) \ c_1) \rightarrow s \ (s \ (Y \ (\lambda f. \lambda x. s \ (f \ x)) \ c_1)) \rightarrow \dots$$

## 4.5 評価戦略

ラムダ計算は計算機械ではない。なぜかといえば、簡約という計算機構があっても、それをどの順番で行うか、あるいはいつ計算が終わるか、が定まっていない。標準形と戦略はちょうどそれを決める役割をはたしている。

**標準形** 簡約できる個所を全く残さない(既約)標準形以外にも、先頭の部分だけが簡約できない弱冠頭標準形も定義できる。例えば  $(x \ ((\lambda y. y) \ z))$  や  $\lambda x. ((\lambda y. y) \ z)$  は既約標準形でない ( $y$  がまだ簡約できる) が弱冠頭標準形である。形式化すると、 $\lambda x. M$  または  $(x \ M_1 \ \dots \ M_n)$  のどちらかの形をしたものが弱冠頭標準形である。

**最左戦略** 最も左にある簡約できる個所を先に簡約する。名前呼び出し (call-by-name) に当る。関数の引数を評価せずに、そのまま代入する。 $(\lambda x. x) \ ((\lambda y. y) \ z) \rightarrow ((\lambda y. y) \ z)$ 。ある戦略で  $M \rightarrow^* N$  ( $N$  標準形) のとき、最左戦略でも  $M \rightarrow^* N$ 。

**最右最内戦略** 最も右にある簡約できる個所の中で、最も中にあるものを選ぶ。値呼び出し (call-by-value) に当る。関数の引数を標準形に落してから代入する。 $(\lambda x. x) \ ((\lambda y. y) \ z) \rightarrow ((\lambda x. x) \ z)$ 。ある戦略で  $M \rightarrow^* \dots$  (無限に続く簡約) のとき、最右最内戦略でも無限な簡約になる。

**$\lambda$  機械** 標準形と戦略の組み合わせを選ぶと、任意の  $\lambda$  項をある  $\lambda$  機械によって決定的に実行されるプログラムとしてみるができる。