untyped λ-calculus

myuon

2018 年 9 月 1 日

目次

1 Syntax

1.1 ラムダ項

定義 1. λ 項

λ **項**が次で定義される. また, こうして定義された λ 項の集合を Λ とかく.

$\langle lambda \rangle ::= \langle var \rangle$
| $\lambda \langle var \rangle$ '.' $\langle lambda \rangle$
| $\langle lambda \rangle \langle lambda \rangle$

ただし, $\langle \mathrm{var} \rangle$ とは変数のことであり, ここでは適当な変数の集合 V が与えられているとしている. 変数を以下では $a, b, c, \ldots$ あるいは $v_0, v_1, v_2, \ldots$ などと書く.

上の 2 番目の規則を**関数抽象**, 3 番目の規則を**関数適用**と呼ぶ.

関数抽象とは項に出現する (かもしれない) 変数を 1 つ固定してそれに依存する関数を作る操作であり, 関数適用とは関数を項へ適用[*1]する操作のことである. 例えば 2 乗関数 $f(x) = x^2$ は項 x^2 に出現する変数 x に依存した関数 f を定義している. よってこれは λ 項風にかけば $\lambda x.\ x^2$ であり, この関数を 10 に適用した $f(10)$ は $f\ 10$ である[*2].

関数適用は左結合 $(M_1 M_2 M_3 = (M_1 M_2) M_3)$ であり, 関数抽象よりも結合の優先順位が高い $(\lambda x.\ M_1 M_2 = \lambda x.\ (M_1 M_2))$.

さて, 関数抽象とは関数を定義することであったが, 通常変数の名前の付け替えにはよらない. 平たくいえば,$f(x) = x$ と $f(y) = y$ は通常区別しない. このような名前の付け替えを行う変換のことを, α 変換とよぶ.

[*1] よく間違えられるが, "関数" を "値" へ適用する. 逆ではない.
[*2] ここでは 2 乗や 10 といった値は λ 項ではないので厳密にはこれらは λ 項ではない

定義 2. x, y を変数, M を λ 項とするとき, 次の操作を α **変換**とよぶ.

$$\lambda x.\ M \quad \longrightarrow \quad \lambda y.\ M[y/x]$$

ただし $M[y/x]$ とは項 M に出現する変数 x を全て y に置き換えた項を表す. 正確には, 次のようにして代入が定義される:

$$\begin{aligned} -[-/-] &: \Lambda \times \Lambda \times V \to \Lambda \\ x[N/x] &= N \\ y[N/x] &= y \ (x \neq y) \\ (\lambda y.\ M)[N/x] &= \lambda y.\ M[N/x] \\ (M_1 M_2)[N/x] &= M_1[N/x]\ M_2[N/x] \end{aligned}$$

α 変換から生成される Λ 上の同値関係を α 同値とよび, $=_\alpha$ とかく.

1.2 評価

ここではラムダ項を別のラムダ項に変換することを考える. ラムダ項の変換のことを評価・計算と呼んだりする.

定義 3 (β 変換)**.** β **変換**を次のように定義する.

1. $M \to_\beta N$ のとき, $\lambda x.M \to_\beta \lambda x.N$
2. $M \to_\beta N$ のとき, $ML \to_\beta NL$, $LM \to_\beta LN$
3. $(\lambda x.M)N \to_\beta M[N/x]$

4 番目の規則のことを β 簡約と呼ぶ.

1.3 値と評価戦略

ラムダ計算をプログラミング言語とみなす立場では, プログラムの実行に相当する, 与えられたラムダ項を別のラムダ項に変換する決定的な手続きが必要になる. ところでラムダ項の評価は基本的に前述の β 簡約を基本とするが, 一般にラムダ項には複数の β 基が含まれており, それらをどの順番で評価するかによって結果は異なってくる. このようなラムダ項の簡約の手順のことを評価戦略とよぶ.

以前の β 変換では, $M \to_\beta N$ が M の “いずれかの”β 基を変換することで N が得られる, ということを表していたのに対し, ここで考えるような評価戦略では実際にどの β 基を変換するかまで常に指定されている, という違いがある.

1.3.1 値

定義 4 (値)**.** 値の集合 Val を次のように定義する.

1. $M \in \Lambda \Longrightarrow \lambda x.M \in \mathrm{Val}$

2 Operational Semantics

2.1 call-by-value

call-by-value は値呼びと呼ばれる. また, 一番外側のラムダ項を左から順に簡約するので leftmost-outermost reduction とも呼ばれる.

定義 5 (call-by-value)**.**

1. $(\lambda x.M)v \to_{\mathrm{cbv}} M[v/x]$
2. $M \to_{\mathrm{cbv}} N$ のとき, $vM \to_{\mathrm{cbv}} vN$
3. $M \to_{\mathrm{cbv}} N$ のとき, $ML \to_{\mathrm{cbv}} NL$

今は関数抽象を値と見ているので, 関数抽象はそれ以上簡約せずそのままにしておくことに注意.

あるいは, これは evaluation context を用いて定義することもできる.

定義 6 (call-by-value)**. evaluation context** を次とする:

$$C = [] \mid (C\, e) \mid (v\, C)$$

e はラムダ項, v は値 (ここでは関数抽象) とする. [] は穴 (hole) とよばれ, ここに好きなラムダ項を後で挿入できる. これを用いて call-by-value の reduction を次のように定義することができる.

1. $(\lambda x.M)v \to_{\mathrm{cbv}} M[v/x]$
2. $M \to_{\mathrm{cbv}} M'$ のとき, $C[M] \to_{\mathrm{cbv}} C[M']$

evaluation context の 2 番目の規則が $(v\, C)$ となっていることに注意. これによって, 関数適用の項を簡約するときは必ず前から (前にある項が value になっている時) しか簡約できない.