言語モデル論(4)
―λ計算 その1―

米澤　明憲

始めに

- 関数について、持つ個々の数学的領域（型等）に依存しない、関数の一般的な性質を調べる目的
- 1940年代にA. Churchが始めた計算の理論体系
- 作用型プログラミングの基礎
- 式はλ式(λexpression, λterm) と呼ばれ、関数を表す(denoteする)。
- 基本的に文字列の書き換え（簡約）が計算とみなされる体系であるが、その数学的なモデルは、1970年代半ばにD. Scottに構築された。彼はこれでTuring賞。

λ記法

- 通常の関数表記:

  $f(x) > 3$, $f(x) = (x^2+x)^2$, $f(x) = g(x)$

- **$f(x)$は、xという変数を持つ関数を表すのか、関数　f　のxにおける値を表すのか曖昧**
- 同様に、$(x^2+x)^2$という表現もxに$(x^2+x)^2$という数を対応させる(xを変数とする)関数を表しているのか、単に$(x^2+x)^2$という量を表しているのか曖昧なことがある。このため、関数の値と関数自身を区別に明確にする必要あり。
- **この区別を明確にするため: *λ記号*、*λ変数*の導入により名前の無い関数を表記し、対象物として扱える。**

  $\lambda x.f(x)$　　$\lambda x.(x^2+x)^2$

  （プログラムをデータとして扱う。）

λ抽象(λ-abstraction)

- λy.(x^2+2y+1)、これは y を変数とする関数をあらわすが、このなかの x は固定した値と見なされる。
- この式の前にλx.. を付けて: λx.λy.(x^2+2y+1)とすると、 x、y を変数とする関数にすることができる。これを、(yを変数とする) **λ抽象**と呼ぶ。
- 適用(application):
  λ抽象の逆操作、即ちλ抽象した変数の値を固定する。
- (λx.(x^2+x)^2 2)において、x の値を2に固定する操作、あるいは2を適用する操作と、 36が得られる。

  一方

(λx.λy.(y+x) 1)に1を適用するとその結果は λy.(y+1)

普通の数学での関数記法では

- 高階関数 :　　関数twiceを例にとる。

集合Xから集合Yへの関数全体を

$$(X \to Y) = \{f \mid f : X \to Y\}$$

で表す。

いま、集合 $(N \to N)$ から 集合 $(N \to N)$ への
高階関数 twice を次のように定義する

$$twice(f) = \lambda x \in N.\ f(f(x)),\quad (\forall f \in (N \to N)).$$

形式的体系へ

■ 括弧や領域指定の省略:

$$\text{twice} = \lambda f \in (N \to N).\ (\lambda x \in N.f(f(x)))$$

と表される。この例のようにλ記法を多重に組み合わせて使う場合、表現を簡潔にするため括弧を省略して

$$\text{twice} = \lambda f \in (N \to N).\ \lambda x \in N.\ f(f(x))$$

と書く。この式は、twiceに関数 $f \in (N \to N)$ を与えて twice(f) は $\lambda x \in N.\ f(f(x))$ となり、さらにこの関数に $x \in N$ を与えると、(twice(f))(x) = f(f(x)) となることを示している。なお、(twice(f))(x) を略して twice(f)(x) ともかく。

(例えば、twice(suc) (5) = suc(suc(5)) = 7)

さらに、もし前後の文脈から変数fとxの動く範囲が明らかであればそれらも省略して $\text{twice} = \lambda f.\ \lambda x.\ f(f(x))$ と書くこともある。

Curry化（Currying)

- 多変数関数を、高階関数を活用して、多変数関数を1変数のλ記法だけで表現できる。この方法としてcurry化がある。

  $f_{x,y}= \lambda z \in N.\ f(x,y,z)$の例を使ってそのことを説明する。

  いま、xの値を固定しyの値を変化させたとき、自然数yと関数$f_{x,y}$の間の対応関係を示す関数を $f_x : N \rightarrow (N \rightarrow N)$とすると、

  f_xはλ記法で

  $$f_x == \lambda y \in N.f_{x,y} == \lambda y \in N.\lambda z \in N.f(x,y,z)$$

  と表される。この関数f_xは、xの値が決まるとはじめて具体的な関数となるが、自然数xにこの関数f_xを対応させる関数をさらに$f^*: N \rightarrow (N \rightarrow (N \rightarrow N))$とおくと

  $$f^* == \lambda x \in N.\lambda y \in N.\lambda z \in N.f(x,y,z)$$

  である。このとき、例えば

  $$f^*(3) == \lambda y \in N.\lambda z \in N.f(3,y,z)$$
  $$f^*(3)(5) == \lambda z \in N.f(3,5,z)$$
  $$f^*(3)(5)(7) == F(3,5,7).$$

- Curryは米国の論理学者

Curry化とは、より一般的には

任意のN変数関数 $f : X_1 \times X_2 \times \cdots \times X_n \rightarrow X$に対して $f^* = \lambda x_1 \in X_1, \lambda x_2 \in X_2, \cdots \lambda x_n \in X_n . f(x_1, x_2, \cdots, x_n) : X_1 \rightarrow (X_2 \rightarrow \cdots (X_n \rightarrow X) \cdots)$とおくと、高階関数$f^*$とfの間に

$$f^*(x_1)(x_2)\cdots(x_n) = f(x_1, x_2, \cdots, x_n)$$

の関係が成り立つ。言い換えれば、f^*とfは形式が違うが与えられた$x_1, x_2, \cdots, x_n$に対して同じ値を対応させる関数である。このf^*とfを同一視すれば、任意の多変数関数が1変数関数に対するλ記法を使って表されること

このあとλ計算で学ぶこと

- 型なしのλ計算
- Church-Rosser の定理
- 正規化定理
- 算術計算等のλ計算による表現
- *Y*-コンビネータ、再帰プログラムの表現
- 計算可能性
- （型なしλ計算数学的モデルの話、D. Scott）

λ式 (λterm) の定義

■ **定義**: (1) 変数$x_0, x_1, \cdots$はλ式である。

(2) Mがλ式でxが変数のとき(λx.M)はλ式である。

(3) MとNがλ式のとき(MN)はλ式である。

例えば、x,y,f が変数のとき (λx.(f(fx)))や((λf.(fx))(λy.y))はλ式である。

(2)によるλ式の構成(λx.M)を、xに関するMの関数抽象(functional abstraction)、あるいはλ抽象と呼ぶ。

(3)によるλ式の構成(MN)を、MのNに対する関数適用(functional application)とよぶ。

＊ λ式は変数に関数抽象と関数適用を繰り返し行うことによって得られる

■ **省略記法**:

(1) $\lambda x_1 x_2 \cdots x_n.M \equiv (\lambda x_1.(\lambda x_2.(\cdots(\lambda x_n.M)\cdots)))$ $(n \geqq 1)$

(2) $M_1 M_2 M_3 \cdots M_n \equiv ((\cdots((M_1 M_2)M_3)\cdots)M_n)$ $(n \geqq 2)$.

自由変数と束縛変数

- λ式*M*の中に(λ*x*. ・・・)という形に部分式があるとき、*x*はこの部分式の中で束縛(bound)されているという。
- 変数の中で束縛された形で現われる変数を束縛変数。その出現をbound occurrenceと呼ぶ。
- 束縛されない形で現れる変数を自由変数(free variable).その出現をfree occurrenceと呼ぶ。
- *M*の自由変数(出現)全体の集合をFV(*M*)で表す。
- FV(*M*) = ϕ の時、*M* を閉式(closed term)と呼ぶ。
- (λ*x*. ・・・)の中の束縛変数 *x* の全ての出現を完全に新しい(この式に現われない)変数に置き換えた式を、もとの式と同一視する(ことが出来る)。 例えば、

$$\lambda xy.x \equiv \lambda uy.u \equiv \lambda vy.v$$

変数条件

- 一般に、同じλ式の中に、同じ名前の束縛変数と自由変数が出現することは可能。（なぜ？）
- 必要なら、束縛変数を置き換えて、自由変数と束縛変数を異なるようにできる。
- さらに一般に、異なるλ式群

  $M_1, M_2, \ldots, M_n$

  の間で、それらの束縛変数と自由変数が全く重ならないようにすることが常に可能である。
- 自由変数と束縛変数が文字列として同じものがないことを$M_1, M_2, \ldots, M_n$が満たすしているとき、$M_1, M_2, \ldots, M_n$ は変数条件を満たすという。

β 変換

- λ 計算の基本的・唯一の「計算」操作である。
- λ 記法で表された関数を、λ 記法で表されたデータに適用し、関数値を得る操作に対応
- 定義：β 変換 …((λ x. M) N)…の形の λ 式を…(M[x :=N])… に変換する。ただし、…(M[x := N])…は、Mの中の全てのx にNを代入（置換）した結果表す。

  実際、変数条件を満たす M,x,Nに対して、Mの中の全ての(自由な)xをNで置き換えた結果を M[x:=N]とおく。この記法を用いてλ式のβ変換→を次のように再帰的に定義する。

  (1) (λ x.M)N $\underset{\beta}{\rightarrow}$ M[x:=N].

  (2) M $\underset{\beta}{\rightarrow}$ N ならば、λ x.M $\underset{\beta}{\rightarrow}$ λ x.N,

  MP $\underset{\beta}{\rightarrow}$ NP, PM $\underset{\beta}{\rightarrow}$ PN が成立。
- β 変換の対象となる (λ x. M) N の形の λ 式を β 基（β -redex）と呼ぶ。

正規形

- λ式Mがβ基を持たない、即ちこれ以上β変換ができないとき、Mは正規形(normal form)となっているという。
- λ式Mへ変換を繰り返し施し、これ以上β変換が出来なくなるならば、λ式Mは正規形を持つという。
- 但し、全てのλ式Mが正規形を持つわけではない。例えば: $X \equiv \lambda x.xxx$ の時

$$XX \xrightarrow{\beta} XXX \xrightarrow{\beta} XXXX \xrightarrow{\beta} \cdots$$

はXXで始まるただ一つのβ変換列であり、したがってこの式は正規形をもたない。また、正規形を持つλ式でも、変換の仕方によって変換列が無限に続く可能性のあるものもある。

β 変換の繰り返しとその記法

■ 記法：

$P \overset{*}{\underset{\beta}{\rightarrow}} Q$は、次の意味：$P \equiv P_1 \underset{\beta}{\rightarrow} P_2 \underset{\beta}{\rightarrow} \cdots \underset{\beta}{\rightarrow} P_n \equiv Q$ (ただし　$n \geqq 1$)　β 変換可能な λ 式同士を同一視することによって得られる λ 式間の等号関係を $\underset{\beta}{=}$ で表す。すなわち、λ 式P,Qに対して

$$P \equiv P_1 \overset{}{\underset{\beta}{\leftrightarrow}} P_2 \overset{}{\underset{\beta}{\leftrightarrow}} \cdots \overset{}{\underset{\beta}{\leftrightarrow}} P_n \equiv Q$$

(ただし $M \overset{}{\underset{\beta}{\leftrightarrow}} N \overset{\text{def}}{\Leftrightarrow} (M \underset{\beta}{\rightarrow} N$ または $N \underset{\beta}{\rightarrow} M)$)

を満たす $N \geqq 1$と $P_1, P_2, \cdots, P_n$があるとき、$P \underset{\beta}{=} Q$とかく。

β 変換の繰り返しとその記法

■ 例

(1) $(\lambda x_1x_2\cdots x_n.M)N_1N_2\cdots N_n$

$\xrightarrow[\beta]{}$ $(\lambda x_2\cdots x_n.M[x_1:=N_1])N_2\cdots N_n$

$\xrightarrow[\beta]{}$ $(\lambda x_3\cdots x_n.M[x_1:=N_1][x_2:=N_2])N_3\cdots N_n$

$\xrightarrow[\beta]{}$ $\cdots$

$\xrightarrow[\beta]{}$ $M[x_1:=N_1][x_2:=N_2]\cdots[x_n:=N_n].$

特に $(\lambda x_1x_2\cdots x_n.M)x_1x_2\cdots x_n \twoheadrightarrow_\beta M$

(2) $I \equiv \lambda x.x$のとき、$IM \equiv (\lambda x.x)M \rightarrow M$

(3) $K \equiv \lambda xy.x$のとき、$KMN \rightarrow (\lambda y.M)N \xrightarrow[\beta]{} M$

(4) $S \equiv \lambda xyz.xz(yz)$のとき

$SPQR \equiv (\lambda xyz.xz(yz))PQR \xrightarrow[\beta]{} PR(QR)$

$S(\lambda x.M)(\lambda x.N) \twoheadrightarrow_\beta \lambda z.(\lambda x.M)z((\lambda x.N)z)$

$\twoheadrightarrow_\beta \lambda z.M[x:=z]N[x:=z] \equiv \lambda x.MN$

また、S, K, Iの間に次の関係がある。

$SKK \twoheadrightarrow_\beta \lambda z.Kz(Kz) \xrightarrow[\beta]{} \lambda z.z \equiv I.$

SK式の普遍性

■ 問

変数とSとKと関数適用のみを使って構成されるλ式をSK式とよぶ（より正確にいうと、SK式を再帰的に次のように定義する。（イ）変数とSとKはそれぞれSK式である。（ロ）X_1とX_2がSK式のとき、(X_1X_2)はSK式である）。

（1）任意のλ式Mに対してSK式Xが存在してX　Mが成り立つことを、次の（1.1）と（1.2）を確かめることによって証明せよ。

（1.1）SK式Xと変数xに対して、Ax,Xを次のように$\xrightarrow{\beta}$再帰的に定義する。

任意にλ式に対して、それにβ変換されるSK式が必ず存在

$$Ax.X \equiv \begin{cases} SKK & (X \equiv x\text{のとき}) \\ KX & (X\text{がx以外の変数かSまたはKのとき}) \\ S(Ax.X_1)(Ax.X_2) & (X \equiv X_1X_2\text{のとき、ただし}X_1\text{と}X_2\text{はSK式}) \end{cases}$$

このときAx.XはSK式で、かつ$Ax.X \xrightarrow[\beta]{} \lambda x.X$を満たす。

（1.2）一般のλ式Mに対して、Mの中に現れる全てのλをAに置き換えた結果をXとすると、Xは$X \xrightarrow[\beta]{} M$を満たすSK式である。

（2）$M \equiv \lambda xy.xy$に対して$X \xrightarrow[\beta]{} M$を満たすSK式Xを求めよ。

不動点演算子

- λ式Mに対して M'≡M'' M'' 但し M'' ≡ λx. M(xx) とすると、M'≡ (λx.M(xx))M''→ M_β(M''M'')≡MM'
- 即ちM'はMの不動点である。

  通常の関数の場合と同様に、X = M_βXを満たすXをMの不動点という。
- Y ≡ λy.(λx.y(xx))(λx.y(xx))

  により定義すると、Yは任意のλ式Mに対して上記の不動点M',即ちMの不動点を対応させる関数を表す。実際

  $$YM \xrightarrow{}_\beta (\lambda x.M(xx))(\lambda x.M(xx)) \equiv M'$$

  であるから

  $$YM =_\beta M' =_\beta MM' =_\beta M(YM)$$

  Mの不動点を求めてくれる！
- この Y をCurryの不動点演算子と呼ぶ。

Turingの不動点演算子

■ 問

Y’≡ XX (ただし X ≡ λxy.y(xxy))のとき、全てのMについてY’M ≡ M(Y’M)が成り立つことを示せ(このY’をTuringの不動点演算子とよぶ)。

β 変換の戦略

- λ 式*M*に β 変換を施すとき、*M*の中に β 基が複数ある時、どの順番で β 変換するかは大きな問題である。

$$(\lambda x.xx)\underline{((\lambda y.y)z)} \xrightarrow{\beta} (\lambda x.xx)z$$

$$\underline{(\lambda x.xx)((\lambda y.y)z)} \xrightarrow{\beta} ((\lambda y.y)z)((\lambda y.y)z).$$

のように、変換結果が変わってくることがある。

- つぎの場合にように、 β 基の選び方に依っては変換列が有限で終わるか不明なことがある。

$$(\lambda x.xx)(\lambda x.xxy) \rightarrow (\lambda_{\beta} x.xxy)(\lambda x.xxy)$$

- 次に変換する β 基の選択を一般に戦略と呼ぶことがある。（戦略によっては同じ λ 式*M*が正規形に到達しないことがある。）

Church-Rosser 定理

- 変換列は収束するか？
- 収束した時、得られるλ式は同じものか？

定理(Church-Rosser/合流性)：

一つのλ式から$\xrightarrow{\beta}$によって得られた二つのλ式は必ず$\xrightarrow{\beta}$によって再び一つの式に合流されることができる。言い換えれば$M \xrightarrow{\beta} M_i$ (i = 1,2)ならば、あるNについて$M_i \xrightarrow{\beta} N$ (i = 1,2)が成り立つ。この定理により、同じλ式Mから始めて有限のステップで止まる(すなわち、β基がなくなりそれ以上変換が続けられなくなる)β変換列がいくつかあれば、それらの最終結果はみな一致することが分かる。

- β変換が無限に続くときでも無限列の途中でいつでも分かれて正規形のほうに合流できるという主張でもある。

正規化定理

- 定理: もしMが正規形をもつならば、常に最も左（かつ最も外側）にあるβ基に注目してβ変換を繰り返してゆけば、その正規形に到達できる。
- このλ式の変換に関する戦略を正規順序による戦略と呼ぶ。
- 正規順序による変換は普通のプログラミング言語のcall-by-nameに相当する。（なぜか？）
- call-by-valueに相当するλ式の変換の順序は、作用的順序と呼ばれる。
- Church-Rosser定理は古典的な結果であるが、当初その証明が複雑なものであったが、70年代にTaiteにより見通しのものが得られた。