情報数学 最終課題

72043913/t20391ks 澤田 開杜

概要

情報数学の授業で学んだラムダ計算について下記にまとめて記す。

テーマを選んだ動機

最近のプログラミング言語では、HaskellやScalaのような関数型言語でなくとも、関数が第一級オブジェクトとなっているような言語が多い。私もこの授業を受ける前までは、λ式というと"無名関数"・"関数オブジェクト"といったような認識でしかいなかった。しかし、学んでみると普通のプログラム言語のような表現力を持っており、奥が深く、とても興味を持ったためである。

ラムダ計算

ラムダ記法

λ式とは、いわば名前の無い関数である。例えば整数と$+$の演算子がすでに定義されているとして、引数を1つ取り、それに1を加える関数を定義するとなると、$add1(x) = x + 1$のように記述するだろう。しかし、関数の名前自体は本質的ではない。そこで、この関数を$\lambda x.x + 1$と表現することにする。"λ"という記号の直後に引数を記述し、"."の後に関数の本体を記述する。このように仮引数を指定し、式から直接関数を定義する操作を**λ抽象**という。このように定義したλ式を$add1(3)$のように関数を値に適用するには、$(\lambda x.x + 1)(3)$のように記述する。

カリー化

先程は$add1$という、引数が一つの関数をもとにλ式での形を考えた。ここで新しく$add(x, y) = x + y$という2つの引数を取る関数を考え、それをλ抽象した結果を考えるが、λ式では、一般に$\lambda(x, y).x + y$のような複数引数を受け取る形では記述しない。そのため、これをλ抽象するには**カリー化**という手法が大切になる。まずは普通の関数をカリー化するとはどういうことかを考える。関数addは2つの数を受け取って1つの数を返すため、型は$N \times N \to N$である。一言で説明すると、カリー化とはこの$N \times N \to N$のような関数を$N \to N \to N$のような形にすることである。カリー化したaddをadd'と表記すると、add'の例の場合、引数を1つ与えると**"引数を1つ受け取り、1つの値を返す$N \to N$型の関数"**を返す。つまり、$add'(1)$とすると先程の$add1$と同じ、引数を1つ受け取り、それに1を足した結果を返す関数を作ることができる。よって、addに対して$add(1)(7)$のように呼び出せば$1 + 7$を計算することができる。これをλ式では$\lambda x.(\lambda y.x + y)$と記述する。更に、λ式では曖昧性がなければカッコやλを省略することができるため、$\lambda x\ y.x + y$と記述するのが一般的である。また、引数に関数を取ったり、戻り値として関数を返したりするような関数のことを、一般に**高階関数**という。この高階関数はラムダ計算だけでなく、最近のプログラム言語では機能としてサポートされている物が多い。具体れは示さないが、これをうまく使えると美しいプログラムを書くことができる。

λ式の定義とβ変換

λ式を以下のように定義する。

1. 変数$x, y, z, x_1, x_2, y', ...$はλ式である
2. λ式Mと変数xに対して、$(\lambda x. M)$はλ式である。(**λ抽象**, $function\ abstraction$)
3. λ式MとNに対して$(M\ N)$はλ式である。(**関数適用**, $function\ appliacation$)

λ抽象については前の *ラムダ記法* の部分で記したが、関数適用については深く記していない。しかし、難しいことは全くなく、 *カリー化* の部分で述べた2つの数を足し算をするλ式で考えると、$(\lambda x\ y. x + y)(1)(7)$とするとλ式に1を適用し、後に7を適用している状態になる。この計算の過程を詳しく確認すると、まず$(\lambda x\ y. x + y)(1)(7)$の$x$が1に置き換わり、$(\lambda y. 1 + y)(7)$となり、次に$y$が7に置き換わって$1 + 7$となる。そして、この**仮引数に実引数に置き換える**という手順のことをβ**変換**といい、以降"$\overset{\beta}{\rightarrow}$"と記す。$(\lambda x. M)N$のように$\beta$変換が可能な形になっている部分のことを$\beta$基といい、逆に$\beta$基を持たない$\lambda x\ y. x\ y$のようなλ式のことを**正規形**($normal\ form$)という。また、**カリー化**の部分でも記したが、λ式には以下のような省略表現が存在する。

1. $\lambda x_1\ x_2\ ...x_n. M \equiv (\lambda x_1. (\lambda x_2. (...(\lambda x_n. M)...)))$
2. $M_1\ M_2\ M_3\ ...M_n \equiv ((...((M_1\ M_2)M_3)...)M_n)$

束縛変数と自由変数

λ式の中で変数xが$\lambda x. --x--$のように現れた時、「変数xはλxによって束縛されている」といい、これを**束縛変数**という。逆に、束縛されていない変数のことを**自由変数**という。また、自由変数を持たないλ式のことを**閉式**という。λ式M中の自由変数の全体を$FV(M)$で表すとき、閉式であるときは$FV(M) = \emptyset$である。

α変換と代入

束縛された変数を置き換えても意味は変わらない。例えば、$\lambda x. x$と$\lambda y. y$は同じである。このように束縛された変数を置き換える操作のことをα**変換**といい、以降、"$\overset{\alpha}{\rightarrow}$"とする。$\alpha$変換を使うことによって、自由変数と束縛変数が異なるようにすることができる。また、式Mの自由変数xを式Nに置き換えることを**代入**といい、以降"$M[x := N]$"とする。具体例としては、$((\lambda x. y\ x)y)[y := (\lambda z. z)] \equiv (\lambda x. (\lambda z. z)x)(\lambda z. z)$となる。ただし、変数の束縛関係を変えてはいけないため、$((\lambda x. y\ x)y)[y := x] \not\equiv (\lambda x. x\ x)x$である。この様な場合のときは、$(\lambda x. y\ x)y \overset{\alpha}{\rightarrow} (\lambda z. y\ z)y$のように、$\alpha$変換してかぶっている変数名を変えるなどする必要がある。

最左β変換と最右β変換

最も左にあるβ基を順番に簡約することを**最左**β**変換**といい、最左β変換を繰り返し適用すれば正規形に到達するという特徴がある。ただし、最左β変換が最も効率の良い変換方法であるとは限らず、次に説明する最右β変換の方が効率的である場合もある(むしろ最右β変換のほうが効率的であることのほうが多い)。最左とは逆に、最も右にあるβ基を順番に変換することを**最右**β**変換**といい、正規形にたどり着けないかも知れないが最左β変換に比べて少ない手数で正規形にたどり着ける可能性が高い。以下に同じλ式を最左β変換で簡略した場合と最右β変換で簡略した場合を示す。

λ式$((\lambda x. (xx))((\lambda y. y)z))$について、

- 最左β変換の場合:

$$(\lambda \underline{x}. xx)\underline{((\lambda y. y)z)} \longrightarrow_\beta (\lambda \underline{y}. y)\underline{z}((\lambda y. y)z) \longrightarrow_\beta z((\lambda \underline{y}. y)\underline{z}) \longrightarrow_\beta zz\ (normal\ form)$$

- 最右β変換の場合:

$$(\lambda x. xx)((\lambda \underline{y}. y)\underline{z}) \longrightarrow_\beta (\lambda \underline{x}. xx)\underline{z} \longrightarrow_\beta zz\ (normal\ form)$$

となる。最左β変換

の場合と最右β変換の場合とで同じ正規形にたどり着いたのは偶然ではなく、α変換を除き、λ式の正規形はβ基の選択に関係なく一意的に決まることが一般的に知られている。証明は省略するが、これを$Church-Rosser$**の定理**という。

様々なλ表現

ラムダ記法 、 *カリー化* 、 *λ式の定義とカリー化* の章では勝手に整数や"+"の演算子を使っていたが、本来ラムダ計算には足し算や掛け算などの基本的な演算は愚か、整数すらも定義されていない。底にあるのはλ抽象、関数適用、そして$\alpha\beta$変換だけである。しかし、数値をλ式を使って仮に表現する方法をとり、更にそこに演算を定義すれば、実際のプログラミング言語のように様々な計算や条件分岐などを表現することができる。λ式を使って数値を表現する方法はいくつかあるが、ここでは$A.Church$が考案したチャーチ数という呼ばれるものを使う。以下にその例を示す。

- $[0] \equiv \lambda x\ y.\ y,\ [1] \equiv \lambda x\ y.\ x\ y,\ [2] \equiv \lambda x\ y.\ x(x\ y),\ [3] \equiv \lambda x\ y.\ x(x\ (x\ y)),\ \dots,\ [n] \equiv \lambda x\ y.\ x(x(\dots(x\ y)\dots))$

以上を見ての通り、チャーチ数は**"λ式の本体の中に現れるxの数$=$その自然数"**として定義しているのである。また、真偽値$[true],\ [false]$を以下のように定義する。

- $[true] \equiv \lambda x\ y.\ x,\ [false] \equiv \lambda x\ y.\ y$

そしてこれらの数に適用することのできる、真偽値を用いて定義できる演算を以下のように定義する。

ラムダ表現	λ式
$[suc]$	$\lambda x\ y\ z.\ y(x\ y\ z)$
$[add]$	$\lambda x\ y\ z\ w.\ x\ z(y\ z\ w)$
$[mul]$	$\lambda x\ y\ z\ w.\ x(y\ z)w$
$[pred]$	$\lambda x\ y\ z.\ x(\lambda u\ v.\ v(u\ y))(\lambda a.\ z)(\lambda a.\ a)$
$[iz\_zero]$	$\lambda x.\ x(\lambda x.\ [false])[true]$
$[\pi_1]$	$\lambda x.\ x[true]$
$[\pi_2]$	$\lambda x.\ x[false]$

例として、これらを用いて$[suc][2]$を計算すると、

$$[suc][2] \equiv (\lambda \underline{x}\ y\ z.\ y(x\ y\ z))\underline{(\lambda x\ y.\ x(x\ y))} \longrightarrow_\beta \lambda y\ z.\ y((\lambda \underline{x}\ y.\ x(x\ y))\underline{y}\ z) \longrightarrow_{\alpha\beta} \lambda y\ z.\ y((\lambda \underline{a}.\ y(y\ a))\underline{z}) \longrightarrow_\beta \lambda y\ z.\ y(y(y\ z)) \longrightarrow_\alpha \lambda x\ y.\ x(x(x\ y)) = 3$$

となる。$[add][3][2]$の場合は、

$$\begin{aligned}
&[add][3][2] \equiv (\lambda \underline{x}\ y\ z\ w.\ x\ z(y\ z\ w))\underline{(\lambda x\ y.\ x(x(x\ y)))}(\lambda x\ y.\ x(x\ y)) \\
\longrightarrow_\beta\ & (\lambda \underline{y}\ z\ w.\ (\lambda x\ y.\ x(x(x\ y)))z(y\ z\ w))\underline{(\lambda x\ y.\ x(x\ y))} \\
\longrightarrow_\beta\ & \lambda z\ w.\ (\lambda \underline{x}\ y.\ x(x(x\ y)))\underline{z}((\lambda x\ y.\ x(x\ y))z\ w) \\
\longrightarrow_\beta\ & \lambda z\ w.\ (\lambda \underline{y}.\ z(z(z\ y)))\underline{((\lambda xy.\ x(x\ y))z\ w)} \\
\longrightarrow_\beta\ & \lambda z\ w.\ z(z(z((\lambda \underline{x}\ y.\ x(x\ y))\underline{z}\ w))) \\
\longrightarrow_\beta\ & \lambda z\ w.\ z(z(z((\lambda \underline{y}.\ z(z\ y))\underline{w}))) \\
\longrightarrow_\beta\ & \lambda z\ w.\ z(z(z(z(z\ w)))) \\
\longrightarrow_\alpha\ & \lambda x\ y.\ x(x(x(x(x\ y)))) \\
=\ & 5
\end{aligned}$$

となる。

λ式の再帰的定義

λ式の再帰的定義の前に、普通の関数に置いての再帰的定義を考えてみる。例えば、$n!$を計算する関数$fact$は$fact(n) = if\ n = 0\ then\ 1\ else\ n * fact(n-1)$となるが、$fact$の定義中に$fact$が現れているので、これでは定義が不完全である。ここで、高階関数Fを$F(f)(n) = if\ n = 0\ then\ 1\ else\ n * fact(n-1)$とすると、$fact$は$F(f) = f$の等式を満たす関数$f$と考えられる。この様な等式を満たす$f$のことを$F$の不動点といい、λ計算では任意のλ式について不動点が存在することが知られている。これは**不動点演算子**という特別なλ式を使うことで定義することができる。自然数のλ表現と同様にこれにも様々な種類があるが、ここでは$Curry$の不動点演算子を使うものとする。$Curry$の不動点演算子は$Y \equiv \lambda y.(\lambda x.y(x\ x))(\lambda x.y(x\ x))$と定義される。これと上で示した数値表現、演算を用いることで、階乗を求めるλ表現を定義することができる。

まず、$[f] \equiv \lambda\, n.\,((([isZero]\ n)\ 1)\ (([mul]\ n)\ (f\ ([pred]\ n)))))$とする。すると先ほどと同じように両辺に$f$が出現しており不適切であるため、これを$[H_{fact}] \equiv \lambda\, n.\,((([isZero]\ n)\ 1)\ (([mul]\ n)\ (f\ ([pred]\ n)))))$とする。あとはこれに$Curry$の不動点演算子を用いて$[fact] \equiv (Y\ H_{fact})$とすれば階乗のλ表現を定義することができる。

まとめ

このようにラムダ計算自体はとても単純なものだが、これのみを用いて自然数を定義して階乗を求めるλ表現も作ることも可能である。これを勉強したからといって今すぐに実践的な美しいコードを書けるようになるわけではないが、情報学的な新しい自然数の定義を学んだり、高階関数を多用する感覚を掴むことは情報を学ぶものとしての嗜みであると私は考える。この授業では取り扱わなかったし、私もまだ勉強中だが、型付きラムダ計算という分野も存在し、それもまたプログラミング言語の型理論と密接に結びついていてとても興味深い。プログラミングだけでなく、プログラミング言語自体が大好きな後輩諸君は、ぜひこの分野を学んでみることを推奨する。

参考文献

- プログラム意味論: [[著]: 横山 寛文, [発行]: 共立出版株式会社]
- 第2章 ラムダ計算（λ-calculus）: [[著]: 香川 考司, [URL]: http://guppy.eng.kagawa-u.ac.jp/~kagawa/2012/AdvProg/Text/Lambda.pdf]
- 計算と論理: [[著]: Jacques Garrigue, [URL]: https://www.math.nagoya-u.ac.jp/~garrigue/lecture/2015_agora/lambda.pdf]
- ラムダ計算処理系ツール: [[作者GitHub]: https://github.com/nikosai , [URL]: https://nikosai.ml/lambda-friends/] ※レポート中のラムダ計算の結果の一部はこのツールを使用して出力しました