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

ラムダ式とは、いわば名前の無い関数である。例えば$+$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ラムダ式での形を考えた。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、$(\lambda x\ y. x + y)(1)(7)$とするとラムダ式に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"$\stackrel{\beta}{\to}$"と記す。$(\lambda x. M)N$のようにβ変換が可能な形になっている部分のことをβ基といい、逆にβ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、ラムダ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、"$\stackrel{\alpha}{\to}$"とする。α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、$(\lambda x. y\ x)y \stackrel{\alpha}{\to} (\lambda z. y\ z)y$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: $((\lambda x. (xx))((\lambda y. y)z)) \stackrel{\beta}{\to} (\lambda y. y)z((\lambda y. y)z) \stackrel{\beta}{\to} z((\lambda y. y)z) \stackrel{\beta}{\to} zz\ (normal\ form)$
- 最右β変換の場合: $((\lambda x. (xx))((\lambda y. y)z)) \stackrel{\beta}{\to} (\lambda x. x)z \stackrel{\beta}{\to} zz\ (normal\ form)$

となる。