5 : The Untyped Lambda-Calculus

大根田裕一
oyuichi@yl.is.s.u-tokyo.ac.jp

Lambda Calculus は、1920 年代、Alonzo Church によって考えられた計算体系である。

1 Basics

Lambda Calculus では、関数適用、関数定義という概念が可能な限り単純な形で表現されている。Lambda Calculus における項 (term) は、次のように帰納的に定義される。

$$\begin{array}{rcll} \mathsf{t}\ (\mathsf{term}) & ::= & \mathsf{x} & (\mathsf{variable}) \\ & | & \lambda \mathsf{x.t} & (\mathsf{abstraction}) \\ & | & \mathsf{t\ t} & (\mathsf{application}) \end{array}$$

Lambda Calculus における唯一の計算は、関数適用の際に起こる。

$$(\lambda \mathsf{x.t_1})\ \mathsf{t_2}$$

のような関数適用が

$$[\mathsf{x} \mapsto \mathsf{t_2}]\mathsf{t_1}$$

と書き換わることが、計算の 1 ステップである。上の項 $[\mathsf{x} \mapsto \mathsf{t_2}]\mathsf{t_1}$ は、`t1` に含まれている変数 `x` を全て `t2` に置き換えた項である。$(\lambda \mathsf{x.t_1})\ \mathsf{t_2}$ という形のした項は *redex* と呼ばれる。redex を上で述べたような

$$(\lambda \mathsf{x.t_1})\ \mathsf{t_2} \ \rightarrow\ [\mathsf{x} \mapsto \mathsf{t_2}]\mathsf{t_1}$$

というルールで書き換えていくことを β **簡約**という。従って、β 簡約を項に対して繰り返し施していくことが Lambda Calculus における計算に他ならない。

例 1.1

$$(\lambda \mathsf{x.\ x}\ (\lambda \mathsf{x.x}))\ (\mathsf{u\ r}) \ \rightarrow\ \mathsf{u\ r}\ (\lambda \mathsf{x.x})$$

一般に Lambda 項は、簡約できる箇所を複数持っている。例えば次の項。

$$(\lambda \mathsf{x.x})\ ((\lambda \mathsf{x.x})\ (\lambda \mathsf{z.}(\lambda \mathsf{x.x})\ \mathsf{z}))$$

複数ある redex をどのような順番で簡約していくか、に関していくつかの戦略がある。すぐ上の例を元に、それらを見てみる。(以下の例では、$\lambda x.x$ を id と表すこととする)

- *full beta-reduction* : 任意の redex が任意の時に簡約できる戦略。例えば次のような簡約ができる。

$$\begin{array}{rl} & \mathsf{id}\ (\mathsf{id}\ (\lambda \mathsf{z.}\ \underline{\mathsf{id}\ \mathsf{z}})) \\ \rightarrow & \mathsf{id}\ \underline{(\mathsf{id}\ (\lambda \mathsf{z.}\ \mathsf{z}))} \\ \rightarrow & \underline{\mathsf{id}\ (\lambda \mathsf{z.}\ \mathsf{z})} \\ \rightarrow & \lambda \mathsf{z.}\ \mathsf{z} \end{array}$$

- *normal order strategy* : 最も左側 ($\approx$ 最も外側) から順に簡約していく方法。

$$\begin{array}{rl} & \underline{\mathsf{id}\ (\mathsf{id}\ (\lambda \mathsf{z.}\ \mathsf{id}\ \mathsf{z}))} \\ \rightarrow & \underline{\mathsf{id}\ (\lambda \mathsf{z.}\ \mathsf{id}\ \mathsf{z})} \\ \rightarrow & \lambda \mathsf{z.}\ \underline{\mathsf{id}\ \mathsf{z}} \\ \rightarrow & \lambda \mathsf{z.z} \end{array}$$

- *call by name* : normal order strategy と同様、最も左側から順に簡約していくが、Lambda 抽象の中は簡約しない。

$$\begin{aligned} & \underline{\mathsf{id}\ (\mathsf{id}\ (\lambda \mathsf{z}.\ \mathsf{id}\ \mathsf{z}))} \\ \rightarrow\ & \underline{\mathsf{id}\ (\lambda \mathsf{z}.\ \mathsf{id}\ \mathsf{z})} \\ \rightarrow\ & \lambda \mathsf{z}.\ \underline{\mathsf{id}\ \mathsf{z}} \end{aligned}$$

  途中までは normal order strategy と同じだが、最後の式 $\lambda z.\ \underline{id\ z}$ の下線部は Lambda 抽象の中なので、簡約されない。このような簡約方法は Haskell や Algol-60 の評価方法の元となっている。(実際、Haskell は call by name よりもっと効率的な評価方法、call by need を採用している。call by need では引数が評価されるとき、その評価結果をどこかに記録しておく。そして以後の計算で同一の引数の評価が起こる際、再び評価を行うようなことはせずに、先ほど記録した評価結果をそのまま使う。)

- *call by value* : call by name に「引数を先に簡約してしまう」というさらなる制限を加えた戦略。ほとんどのプログラミング言語の評価方法は、この簡約戦略に基づいている。

$$\begin{aligned} & \mathsf{id}\ \underline{(\mathsf{id}\ (\lambda \mathsf{z}.\ \mathsf{id}\ \mathsf{z}))} \\ \rightarrow\ & \underline{\mathsf{id}\ (\lambda \mathsf{z}.\ \mathsf{id}\ \mathsf{z})} \\ \rightarrow\ & \lambda \mathsf{z}.\ \mathsf{id}\ \mathsf{z} \end{aligned}$$

2 Programming in the Lambda-Calculus

第 1 章で見たように Lambda Calculus のシンタックスは非常に単純なものであるが、通常のプログラミング言語によくある複雑なデータ構造やシンタックスと等価なものを Lambda Calculus 内で表現することが出来る。

2.1 複数の引数

複数の引数を受け取るような関数は、高階関数を利用することで実現できる。$\mathsf{f} = \lambda(\mathsf{x},\mathsf{y}).\mathsf{s}$ というような関数は、$\mathsf{f} = \lambda \mathsf{x}.\ \lambda \mathsf{y}.\ \mathsf{s}$ と同じことである。このように、複数の引数を持つ関数を複数の単一引数の関数に変換することを、カリー化 (currying) という。

2.2 真偽値, and, or, not, if 分岐

まず真偽値 (true, false) を次のように定める。

$$\begin{aligned} \mathsf{true} &:= \lambda \mathsf{t}.\lambda \mathsf{f}.\ \mathsf{t} \\ \mathsf{false} &:= \lambda \mathsf{t}.\lambda \mathsf{f}.\ \mathsf{f} \end{aligned}$$

すると、if 分岐は次のように表すことが出来る。

$$\mathsf{if} := \lambda \mathsf{l}.\lambda \mathsf{m}.\lambda \mathsf{n}.\ \mathsf{l}\ \mathsf{m}\ \mathsf{n}$$

一つ目の引数として、上で定義した真偽値 (true, false) を与える。2 つ目 3 つ目の引数としてそれぞれ、l が真のときに返す項、が偽のときに返す項を与える。true, false の定義により、うまく if 文の機能を果たしていることが分かる。

また、and, or, not はそれぞれ次のような Lambda 項になる。

$$\begin{aligned} \mathsf{and} &:= \lambda \mathsf{b}.\lambda \mathsf{c}.\ \mathsf{b}\ \mathsf{c}\ \mathsf{false} \\ \mathsf{or} &:= \lambda \mathsf{b}.\lambda \mathsf{c}.\ \mathsf{b}\ \mathsf{true}\ \mathsf{c} \\ \mathsf{not} &:= \lambda \mathsf{b}.\ \mathsf{b}\ \mathsf{false}\ \mathsf{true} \end{aligned}$$

2.3 Pair

値のペアも (従ってリストも) 次のような関数を定義することでうまく表現できる。

$$\begin{aligned}\mathsf{pair} &:= \lambda \mathsf{f}.\lambda \mathsf{s}.\lambda \mathsf{b}.\ \mathsf{b}\ \mathsf{f}\ \mathsf{s}\\ \mathsf{fst} &:= \lambda \mathsf{p}.\ \mathsf{p}\ \mathsf{true}\\ \mathsf{snd} &:= \lambda \mathsf{p}.\ \mathsf{p}\ \mathsf{false}\end{aligned}$$

関数 pair の仮引数 f は一つ目の要素に、s は２つ目の要素に対応している。pair からの値の取り出しは fst と snd によって行う。

2.4 自然数

Lambda Calculus 上では、自然数 (0, 1, 2, 3, ...) に対応するものとして、次のような Lambda 項 c_0, c_1, c_2, c_3 ... を考える。

$$\begin{aligned}\mathsf{c}_0 &:= \lambda \mathsf{s}.\lambda \mathsf{z}.\ \mathsf{z}\\ \mathsf{c}_1 &:= \lambda \mathsf{s}.\lambda \mathsf{z}.\ \mathsf{s}\ \mathsf{z}\\ \mathsf{c}_2 &:= \lambda \mathsf{s}.\lambda \mathsf{z}.\ \mathsf{s}\ (\mathsf{s}\ \mathsf{z})\\ \mathsf{c}_3 &:= \lambda \mathsf{s}.\lambda \mathsf{z}.\ \mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z}))\\ &\vdots\end{aligned}$$

自然数 n は２つの引数 (s と z—"successor"と"zero") を受け取り、zero に対し successor—1 を足す関数—を n 回適用する関数として表現される。

successor は次のように定義できる。

$$\mathsf{succ} := \lambda \mathsf{n}.\lambda \mathsf{s}.\lambda \mathsf{z}.\ \mathsf{s}\ (\mathsf{n}\ \mathsf{s}\ \mathsf{z})$$

例 2.1

$$\begin{aligned}\mathsf{succ}\ \mathsf{c}_\mathsf{k} &= (\lambda \mathsf{n}.\lambda \mathsf{s}.\lambda \mathsf{z}.\ \mathsf{s}\ (\mathsf{n}\ \mathsf{s}\ \mathsf{z}))\ (\lambda \mathsf{s}.\lambda \mathsf{z}.\ \underbrace{\mathsf{s}\ \cdots\ (\mathsf{s}}_{k}\ \mathsf{z}))\\ &= \lambda \mathsf{s}.\lambda \mathsf{z}.\ \mathsf{s}\ (\lambda \mathsf{s}.\lambda \mathsf{z}.\ \mathsf{s}\ (\underbrace{\mathsf{s}\ \cdots\ (\mathsf{s}}_{k}\ \mathsf{z}))\ \mathsf{s}\ \mathsf{z}\\ &= \lambda \mathsf{s}.\lambda \mathsf{z}.\ \underbrace{\mathsf{s}\ (\mathsf{s}\ \cdots\ (\mathsf{s}}_{k+1}\ \mathsf{z}))\\ &= \mathsf{c}_{\mathsf{k}+1}\end{aligned}$$

足し算は次のように定義できる。

$$\mathsf{plus} := \lambda \mathsf{m}.\lambda \mathsf{n}.\lambda \mathsf{s}.\lambda \mathsf{z}.\ \mathsf{m}\ \mathsf{s}\ (\mathsf{n}\ \mathsf{s}\ \mathsf{z})$$

例 2.2

$$\begin{aligned}\mathsf{plus}\ \mathsf{c}_\mathsf{m}\ \mathsf{c}_\mathsf{n} &= (\lambda \mathsf{m}.\ \lambda \mathsf{n}.\ \lambda \mathsf{s}.\ \lambda \mathsf{z}.\ \mathsf{m}\ \mathsf{s}\ (\mathsf{n}\ \mathsf{s}\ \mathsf{z}))\ \mathsf{c}_\mathsf{m}\ \mathsf{c}_\mathsf{n}\\ &= \lambda \mathsf{s}.\ \lambda \mathsf{z}.\ \mathsf{c}_\mathsf{m}\ \mathsf{s}\ (\mathsf{c}_\mathsf{n}\ \mathsf{s}\ \mathsf{z})\\ &= \lambda \mathsf{s}.\ \lambda \mathsf{z}.\ (\lambda \mathsf{s}.\ \lambda \mathsf{z}.\ \underbrace{\mathsf{s}\ \cdots\ (\mathsf{s}}_{m}\ \mathsf{z}))\ \mathsf{s}\ (\lambda \mathsf{s}.\ \lambda \mathsf{z}.\ \underbrace{\mathsf{s}\ \cdots\ (\mathsf{s}}_{n}\ \mathsf{z}))\\ &= \lambda s.\ \lambda z.\ \underbrace{\mathsf{s}\ \cdots\ (\mathsf{s}}_{m+n}\ z)\\ &= \mathsf{c}_{\mathsf{m}+\mathsf{n}}\end{aligned}$$

かけ算は次のように定義できる。

$$\mathsf{times} \;:=\; \lambda \mathsf{m}.\lambda \mathsf{n}.\ \mathsf{m}\ (\mathsf{plus}\ \mathsf{n})\ \mathsf{c}_0$$

例 2.3

$$\begin{aligned}
\mathsf{times}\ \mathsf{c}_\mathsf{m}\ \mathsf{c}_\mathsf{n} &= \mathsf{c}_\mathsf{m}\ (\mathsf{plus}\ \mathsf{c}_\mathsf{n})\ \mathsf{c}_0 \\
&= (\lambda \mathsf{s}.\ \lambda \mathsf{z}.\ \underbrace{\mathsf{s}\ \cdots\ (\mathsf{s}}_{m}\ \mathsf{z}))\ (\mathsf{plus}\ \mathsf{c}_\mathsf{n})\ \mathsf{c}_0 \\
&= \underbrace{(\mathsf{plus}\ \mathsf{c}_\mathsf{n})\ \cdots\ ((\mathsf{plus}\ \mathsf{c}_\mathsf{n})}_{m}\ \mathsf{c}_0)
\end{aligned}$$

0 かどうかの判定は次のように定義できる。

$$\mathsf{iszero} \;:=\; \lambda \mathsf{m}.\ \mathsf{m}\ (\lambda \mathsf{x}.\ \mathsf{false})\ \mathsf{true}$$

引き算の定義はやや複雑である。引き算を実現するためには、pair c_n $\mathsf{c}_{\mathsf{n}+1}$ という形のペアを使う必要がある。pair c_n $\mathsf{c}_{\mathsf{n}+1}$ に対する successor(つまり、pair c_n $\mathsf{c}_{\mathsf{n}+1}$ を与えれば pair $\mathsf{c}_{\mathsf{n}+1}$ $\mathsf{c}_{\mathsf{n}+2}$ を返す関数) として、まず

$$\mathsf{ss} \;:=\; \lambda \mathsf{p}.\ \mathsf{pair}\ (\mathsf{snd}\ \mathsf{p})\ (\mathsf{plus}\ \mathsf{c}_1\ (\mathsf{snd}\ \mathsf{p}))$$

を定義する。さらに今、

$$\mathsf{zz} \;:=\; \mathsf{pair}\ \mathsf{c}_0\ \mathsf{c}_0$$

とすると、successor の反対、つまり 1 マイナスする関数は次のようにつくることが出来る。

$$\mathsf{pred} \;:=\; \lambda \mathsf{m}.\ \mathsf{fst}\ (\mathsf{m}\ \mathsf{ss}\ \mathsf{zz})$$

例 2.4

$$\begin{aligned}
\mathsf{pred}\ \mathsf{c}_\mathsf{m} &= \mathsf{fst}\ (\mathsf{c}_\mathsf{m}\ \mathsf{ss}\ \mathsf{zz}) \\
&= \mathsf{fst}\ (\underbrace{\mathsf{ss}\ \cdots\ (\mathsf{ss}}_{m}\ \mathsf{zz})) \\
&= \mathsf{fst}\ (\mathsf{pair}\ \mathsf{c}_{\mathsf{m}-1}\ \mathsf{c}_\mathsf{m}) \\
&= \mathsf{c}_{\mathsf{m}-1}
\end{aligned}$$

pred を用いて、引き算は次のように定義できる。

$$\mathsf{minus} \;:=\; \lambda \mathsf{m}.\ \lambda \mathsf{n}.\ \mathsf{n}\ \mathsf{pred}\ \mathsf{m}$$

2.5 再帰関数

Lambda 項で再帰関数を定義することを考える。例として自然数の階乗を求める関数を考えよう。

$$\mathsf{fact} \;:=\; \lambda \mathsf{n}.\ \mathsf{if}\ (\mathsf{iszero}\ \mathsf{n})\ \mathsf{c}_0\ (\mathsf{times}\ \mathsf{n}\ (\mathsf{fact}\ (\mathsf{pred}\ \mathsf{n})))$$

fact は上の式を満たすべきなのは明らかであるが、関数 fact の Lambda 項による定義にはなってない。右辺の fact は今まさに定義しようとしている Lambda 項だからである。

今仮に

$$\mathsf{g} \;:=\; \lambda \mathsf{fct}.\lambda \mathsf{n}.\ \mathsf{if}\ (\mathsf{iszero}\ \mathsf{n})\ \mathsf{c}_0\ (\mathsf{times}\ \mathsf{n}\ (\mathsf{fct}\ (\mathsf{pred}\ \mathsf{n})))$$

という関数を考えてみる。このとき、「簡約していけば、g に対し自分自身を適用するという形になる」Lambda 項、つまり、

$$\hat{\mathsf{t}} \;\rightarrow^*\; \mathsf{g}\ \hat{\mathsf{t}}$$

という形の Lambda 項 $\hat{t}$ を g に対して適用してやれば、所望の fact が得られる。
このような $\hat{t}$ をどのようにして作るのか？　それには、まず次のような Lambda 抽象を考えればよい。

$$\mathsf{fix} \;:=\; \lambda \mathsf{f}.\ (\lambda \mathsf{x}.\ \mathsf{f}\ (\lambda \mathsf{y}.\ \mathsf{x}\ \mathsf{x}\ \mathsf{y}))\ (\lambda \mathsf{x}.\ \mathsf{f}\ (\lambda \mathsf{y}.\ \mathsf{x}\ \mathsf{x}\ \mathsf{y}))$$

これに g を適用してみると、

$$\begin{aligned}
\mathsf{fix\ g} &\rightarrow (\lambda \mathsf{f}.\ (\lambda \mathsf{x}.\ \mathsf{f}\ (\lambda \mathsf{y}.\ \mathsf{x}\ \mathsf{x}\ \mathsf{y}))\ (\lambda \mathsf{x}.\ \mathsf{f}\ (\lambda \mathsf{y}.\ \mathsf{x}\ \mathsf{x}\ \mathsf{y})))\ \mathsf{g} \\
&\rightarrow (\lambda \mathsf{x}.\ \mathsf{g}\ (\lambda \mathsf{y}.\ \mathsf{x}\ \mathsf{x}\ \mathsf{y}))\ (\lambda \mathsf{x}.\ \mathsf{g}\ (\lambda \mathsf{y}.\ \mathsf{x}\ \mathsf{x}\ \mathsf{y})) \\
&\rightarrow \mathsf{g}\ (\lambda \mathsf{y}.\ \mathsf{h}\ \mathsf{h}\ \mathsf{y}) \\
&\qquad\quad (\mathsf{h} \;:=\; \lambda \mathsf{x}.\ \mathsf{g}\ (\lambda \mathsf{y}.\ \mathsf{x}\ \mathsf{x}\ \mathsf{y})) \\
&\rightarrow \mathsf{g}\ (\lambda \mathsf{y}.\ (\mathsf{fix\ g})\ \mathsf{y})
\end{aligned}$$

となる。call by value の場合、fix g と λy. (fix g) y は等価であること [1]を考えると、この fix g が今求めている $\hat{t}$ であるといえる。従って、求めるべき fact は (call by value では) 次のように書ける。

$$\mathsf{fact} \;:=\; \mathsf{g}\ (\mathsf{fix\ g}) \;=\; \mathsf{fix\ g}$$

3 Formalities

Lambda Calculus の syntax と operational semantics を定義していく。

3.1 Syntax

Definition 3.1 (TERMS) $\mathcal{V}$ を変数の名前の可算な集合とする。この時、*TERMS* の集合とは以下を満たす最小の集合 $\mathcal{T}$ である。

1. $\mathsf{x} \in \mathcal{V}$ ならば $\mathsf{x} \in \mathcal{T}$
2. $\mathsf{t}_1 \in \mathcal{T}$ かつ $\mathsf{x} \in \mathcal{T}$ ならば、$\lambda \mathsf{x}.\mathsf{t}_1 \in \mathcal{T}$
3. $\mathsf{t}_1 \in \mathcal{T}$ かつ $\mathsf{t}_2 \in \mathcal{T}$ ならば、$\mathsf{t}_1\ \mathsf{t}_2 \in \mathcal{T}$

これから先、変数の名前は一意であると仮定する (α 変換されていると仮定する)。

Definition 3.2 項 t の自由変数（$FV(\mathsf{t})$ とする）は次のように定義される。

$$\begin{aligned}
FV(\mathsf{x}) &= \{\mathsf{x}\} \\
FV(\lambda \mathsf{x}.\mathsf{t}_2) &= FV(\mathsf{t}_1) - \{\mathsf{x}\} \\
FV(\mathsf{t}_1\ \mathsf{t}_2) &= FV(\mathsf{t}_1) \cup FV(\mathsf{t}_2)
\end{aligned}$$

3.2 Substitution

Definition 3.3 (SUBSTITUTION)

$$\begin{aligned}
[\mathsf{x} \mapsto \mathsf{s}]\mathsf{x} &= \mathsf{s} \\
[\mathsf{x} \mapsto \mathsf{s}]\mathsf{y} &= \mathsf{y} \\
[\mathsf{x} \mapsto \mathsf{s}](\lambda \mathsf{y}.\mathsf{t}_1) &= \lambda \mathsf{y}.[\mathsf{x} \mapsto \mathsf{s}]\mathsf{t}_1 \\
[\mathsf{x} \mapsto \mathsf{s}](\mathsf{t}_1\ \mathsf{t}_2) &= [\mathsf{x} \mapsto \mathsf{s}]\mathsf{t}_1\ [\mathsf{x} \mapsto \mathsf{s}]\mathsf{t}_2
\end{aligned}$$

[1] (fix g) x = (λy. (fix g) y) x であるということ

3.3 Operational Semantics

Definition 3.4

$$\begin{array}{rcll} \mathsf{t}\ (\mathsf{term}) & ::= & \mathsf{x} & (\mathsf{variable}) \\ & | & \lambda \mathsf{x.t} & (\mathsf{abstraction}) \\ & | & \mathsf{t\ t} & (\mathsf{application}) \end{array}$$

Definition 3.5

$$\mathsf{v}\ (\mathsf{value}) \quad ::= \quad \lambda \mathsf{x.t} \quad (\mathsf{abstraction\ value})$$

Definition 3.6

$$\frac{\mathsf{t}_1 \ \rightarrow \ \mathsf{t}_1'}{\mathsf{t}_1\ \mathsf{t}_2 \ \rightarrow \ \mathsf{t}_1'\ \mathsf{t}_2} \ (E-APP1)$$

$$\frac{\mathsf{t}_2 \ \rightarrow \ \mathsf{t}_2'}{\mathsf{v}_1\ \mathsf{t}_2 \ \rightarrow \ \mathsf{v}_1\ \mathsf{t}_2'} \ (E-APP2)$$

$$(\lambda \mathsf{x.t}_1)\ \mathsf{v}_2 \ \rightarrow \ [\mathsf{x} \mapsto \mathsf{v}_2]\mathsf{t}_1 \quad (E-APPABS)$$

$E-APPABS$ ルールの左辺は、右オペランドが既に value に簡約されている application としか、マッチしない。また、$E-APP1$ ルールは、左オペランドがまだ term である application としかマッチしない。その一方で $E-APP2$ ルールは、適用の左オペランドが value になるまでマッチしない。

以上のことは、適用 $(\mathsf{t}_1\ \mathsf{t}_2)$ の際の評価の順番を自然に定めている。

(1) $E-APP1$ により、項 t_1 を value に簡約。

(2) $E-APP2$ により、項 t_2 を value に簡約。

(3) $E-APPABS$ を使って、適用を簡約。