

# ラムダ計算入門

中田 昌輝

2021 年 2 月 6 日

## 目次

1	ラムダ計算	1
1.1	ラムダ計算とは	1
1.2	First-class と Second-class	1
1.3	高階関数	2
1.4	ラムダ計算の目的と意義	3
1.5	ラムダ式 (ラムダ項) の構文	4
1.6	自由変数と束縛変数	5
1.7	ラムダ式の簡約	6
1.7.1	ラムダ計算の small-step semantics	6
1.7.2	評価の行く末	7
1.7.3	ラムダ式の等しさ	8
1.7.4	$\eta$ 簡約と $\eta$ 展開	9
1.8	$\alpha$ 変換	9
1.9	ラムダ式の記述能力	10
1.9.1	ブール値	10
1.9.2	順序対 (tuple)(=直積型の要素)	11
1.9.3	自然数とその算術	12
1.9.4	再帰関数	12
1.10	不動点結合子	13
1.10.1	不動点定理	13
1.10.2	再帰関数の定義	13
1.10.3	再帰関数の意味	14
1.10.4	様々な不動点結合子	14
1.10.5	不動点結合子と Russell のパラドックス	15
1.11	Church-Rosser 定理	15
1.12	標準化定理とリダクション戦略	17
2	型体系 (type systems)	18
2.1	型体系の目的	18
2.2	型体系の意義	18
2.3	型とプログラムの安全性	19
2.4	簡単な関数型言語 FL の構文	20
2.5	FL の型	21
2.6	FL の small-step semantics	21
2.7	形式的体系	22

# 1 ラムダ計算

## 1.1 ラムダ計算とは

ラムダ計算 (lambda calculus) は計算模型のひとつで、計算の実行を関数への引数の評価 (evaluation) と適用 (application) としてモデル化・抽象化した計算体系である。関数を表現する式に文字  $\lambda$  を使うという慣習からこの名前がつけられた。Alan Turing の指導教員である Alonzo Church が 1934 年頃に決定問題 (decision problem, 形式  $s$  言語で記述された命題の真偽を判定する問題) を解くアルゴリズムの存在不存 (Entscheidungsproblem) に関する研究の中で発明したものである。ラムダ計算は主に「計算可能な関数」とは何かを定義するために用いられる。計算の意味論や型理論など、計算機科学のいろいろなところで使われていて、特に LISP, ML, Haskell といった関数型プログラミング言語の理論的基盤として、その誕生に大きな役割を果たした。

## 1.2 First-class と Second-class

プログラミング言語において first-class citizen (first-class object とも言われる) は、生成・代入・(引数・戻り値としての) 受け渡しといったその言語における基本的な操作を制限なしに使用できる対象のことである。たとえば、数学において

$$(a + b) \times (a - b) \quad (1.1)$$

という式があった場合、 $+$ ,  $-$ ,  $\times$  といったものは意味が“固定”されている。 $+$  であれば加算という意味でしか用いられず、一方で加算という意味を定義する必要はない。また、

$$f(x) \quad (1.2)$$

という  $x$  に関する関数  $f$  は、定義すれば  $f$  の意味は固定されるものの、その定義はプログラムによって異なるので“半固定”である。

プログラミングにおいて

$$(\text{if } x > 0 \text{ then } \sin \text{ else } \cos)(y)$$

とかけるかどうかは、関数が一人前に扱われているかどうか依存する。

first-class citizen は、おおよそ以下の性質を満たす (権利を有する) 言語要素のことである。

1. 実行時に作成できる
2. 変数に代入できる
3. 手続きや関数の引数になれる
4. 関数の戻り値になれる

C 言語や C++ では、数値データや構造体は代入で使用でき、関数に渡すこともできる。しかし同じ C/C++ でも配列 (言語組み込みの固定長配列) は、代入することも関数に渡すこともできない。また関数から返すこともできない。配列そのものでなく、ポインタもしくは参照を使って関数の引数や戻り値とすることしかできない。配列の各要素についてはできるが、配列全体をひとつとして扱うことはできない。そのため、C/C++ の配列は、first-class citizen ではない。また、C/C++ の関数も実行時に作成することができないため、first-class citizen ではない。一方で関数ポインタで代用することで配列も引数や戻り値として使用することができるので、関数も同様であるため、C/C++ の関数を second-class citizen という場合がある。

first-class の関数をもたないことの問題点は、数学においても現れている。以下に示す。

1. 関数の定義と関数適用の区別不足

2. 微分の記法 ( ' )
3. 不定積分, 定積分
4. 量子子 (限量子, quantifier),  $\forall, \exists$

微分や量子子は高階関数<sup>\*1</sup>において, 関数 (手続) を引数に取る関数, と考えるのが自然である. たとえば OCaml の `for_all` 関数や, `exists` 関数でいえば

```
val for_all : ('a -> bool) -> 'a list -> bool
val exists : ('a -> bool) -> 'a list -> bool
```

である.

### 1.3 高階関数

これから関数の定義と関数の適用の区別として  $\lambda$  を用いて表現する.

- $\lambda f.f\ 10$   
関数  $f$  をもらって,  $f(10)$  の値を返す高階関数のことである.
- $\lambda f.f(f\ 10)$   
関数  $f$  をもらって,  $f(f(10))$  の値を返す高階関数のことである.
- $\lambda x.(\lambda y.x + y)$   
 $x$  をもらって, 「 $y$  をもらって  $x + y$  を返す関数」を返す高階関数である.
- $\lambda f.(\lambda x.f(f\ x))$   
関数  $f$  をもらって, 「 $x$  をもらって  $f(f(x))$  の値を返す関数」を返す高階関数である.
- $\lambda f.(\lambda g.(\lambda x.g(f\ x)))$   
関数  $f$  をもらって, 「関数  $g$  をもらって, 「 $f$  と  $g$  の合成関数」を返す高階関数」を返す高階関数である.

以下の関数について考えてみる.

$f(x) = x^2 + 2ax + a^2$ $f(a) = 4a^2$	vs.	$f = \lambda x.x^2 + 2ax + a^2$ $f(a) = 4a^2$
--	-----	---

図 1.1 関数の定義と関数適用の区別

図 1.1 から左手は変数  $x$  をもらって, 値  $x^2 + 2ax + a^2$  を返していると考えることができる. 一方で右手は変数  $x$  をもらって, 関数  $x^2 + 2ax + a^2$  を返していると考えることができる.  $f(a)$  に関しては, 左手は値  $a$  を代入して値  $4a^2$  を返しているが, 右手は関数  $f$  をもらって値  $a$  をもらって, 関数  $f(a)$  である  $4a^2$  を返していると考えることができる.

このように考えれば, 不定積分も高階関数とみなすことができる.

図 1.2 に関して, 右手は積分の関数を  $f$  として,  $x$  の関数  $x^2$  を  $g$  とすると, 関数  $g$  をもらって  $f$  を作用させ, その結果として原子関数を返すと表現することができる. 他の高階関数の例を以下に挙げる.

<sup>\*1</sup> higher-order function, first-class の関数をサポートしているプログラミング言語において, 関数 (手続) を引数にとる関数のことである. 主に関数型言語やその背景理論であるラムダ計算において多用される. 数学では汎関数.

$$\begin{array}{c} \int x^2 dx = \frac{x^3}{3} + C \\ \int y^2 dy = \frac{y^3}{3} + C \end{array} \quad \text{vs.} \quad \begin{array}{c} \int \lambda x. x^2 = \lambda. \left( \frac{x^3}{3} + C \right) \\ \int \lambda y. y^2 = \lambda y. \left( \frac{y^3}{3} + C \right) = \lambda x. \left( \frac{x^3}{3} + C \right) \end{array}$$

図 1.2 不定積分は高階関数

$$\begin{array}{c} f(x) = x^2 + 2ax + a^2 \\ f'(x) = 2x + 2a \\ f'(a) = 4a \end{array} \quad \text{vs.} \quad \begin{array}{c} f = \lambda x. x^2 + 2ax + a^2 \\ f' = \lambda x. 2x + 2a \\ f'(a) = 4a \end{array}$$

図 1.3 微分も高階関数

$$\sum_{k \in S} f(k) \quad \text{vs.} \quad \sum_S f$$

図 1.4  $\sum$  も高階関数

$$\begin{array}{c} \exists y (\text{likes}(x, y)) \\ \forall x \exists y (\text{likes}(x, y)) \end{array} \quad \text{vs.} \quad \begin{array}{c} \exists (\lambda y. \text{likes}(x, y)) \\ \forall (\lambda x. \exists (\lambda y. \text{likes}(x, y))) \\ f'(a) = 4a \end{array}$$

図 1.5  $\forall$  や  $\exists$  も高階関数

## 1.4 ラムダ計算の目的と意義

先で上げた例で言えば、 $f(x)$  というのは関数なのかそれとも  $f$  に  $x$  を食わせた結果の値なのかわからない。もし、関数とデータ（自然数や実数など）が、いつでも文脈から区別できるのであれば、このような曖昧さがあっても構わないが、コンピュータ科学では、関数やプログラム自身を扱う関数（高階関数、メタプログラム）が平気で現れてしまう。そのような場合には、文脈からだけでは、関数なのかデータなのかわからない。そのため関数を一人前のものとして扱うことや広義の関数（ $'$ ,  $\int$ ,  $\sum$ ,  $\forall$ ,  $\exists$  など）を統一的に扱う必要が出てくる。そのためにラムダ計算は用いられる。またラムダ計算の表現力は、計算モデルとしてチューニング機械と同等の表現力をもつ。つまりチューニング機械（や再帰関数やその他のありとあらゆる計算モデル）をシミュレートできるし、逆にチューニング機械でラムダ計算をシミュレートすることもできる。このようなラムダ計算に基づく関数型プログラム言語の特徴として、単一代入、参照透明性（referential transparency）や、計算エフェクトが分離されている（ことが多い）、意味論が明快で検証しやすいこと、簡単な割には強力であることが挙げられる。そのためラムダ計算は計算可能関数とデータの統一表現手段を与える。

ラムダ計算は有効範囲（scope）、変数束縛（variable binding）<sup>\*2</sup>のモデル化が可能である。

最後にラムダ計算は引数評価規則（call by xxx）のモデル化が可能である。引数評価規則として関数型言語での主要な方法として3つある。

- 値呼び出し call by value
- 名前呼び出し call by name
- 必要呼び出し call by need

<sup>\*2</sup> 変数の「宣言（定義）」と「使用（変数の値の参照）」の対応関係。静的束縛と動的束縛がある

## 1.5 ラムダ式 (ラムダ項) の構文

ラムダ式における語彙 (アルファベット) は以下のいずれかである.

- $x, y, z, \dots \in Var$  (変数の可算無限集合)
- $\lambda$

構文に関しては以下のように定義される.

$$\begin{aligned}
 M &::= x \\
 &| (\lambda x. M) \quad (\text{abstraction, 抽象}) \\
 &| (M M) \quad (\text{application, 適用})
 \end{aligned} \tag{1.3}$$

ラムダ計算において, 定数と変数の区別はなく,  $\lambda$ , “.”, “(”, “)” を除くすべての記号が変数というカテゴリに属している.

ラムダ式は,  $L, M, N, \dots$  で表す. また 2 種類の等号が存在する.

- $M \equiv N$  同じ形をしている
- $M = N$  等価である

ラムダ式において演算子の優先順位や結合規則と言ったさまざまな慣習を用いることで, 複合的な式の構造を明示的に表現するための括弧でプログラムを汚してしまう必要性が少なくなる. たとえば  $*$  は  $+$  よりも結合がつよいので, 括弧のない式  $1 + 2 * 3$  を抽象構文木として図 1.6 の左のように解釈する.



図 1.6 抽象構文木の解釈の仕方

ゆえに  $1 + 2 + 3$  という式があった場合は,  $1 + (2 + 3)$  と解釈するのではなく,  $(1 + 2) + 3$  と解釈する. つまり関数適用は左結合的であり,

$$M_1 M_2 M_3 \dots M_n = (\dots ((M_1 M_2) M_3) \dots M_n) \tag{1.4}$$

一方で, 関数抽象は右結合的である. つまり

$$\lambda x_1 x_2 \dots x_n. M = (\lambda x_1. (\lambda x_2. \dots (\lambda x_n. M) \dots)) \tag{1.5}$$

となる.

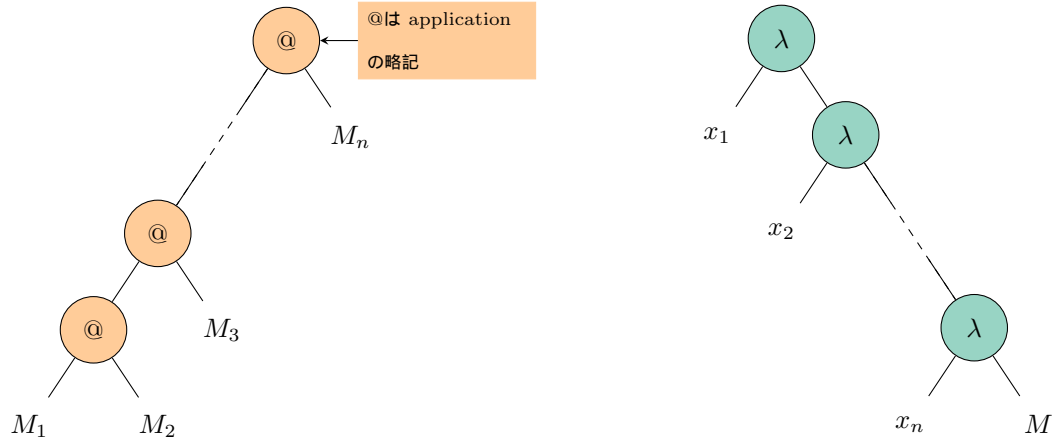


図 1.7 ラムダ式の抽象構文木

ラムダ式の略記法を以下にまとめる。

1.  $(\lambda x_1.(\lambda x_2. \dots (\lambda x_n.M)\dots))$  は  $(\lambda x_1 x_2 \dots x_n.M)$  と略してよい
2.  $(\dots((M_1 M_2) M_3) \dots M_n)$  は  $(M_1 M_2 M_3 \dots M_n)$  と略してよい
3. ラムダ抽象の本体の最外の括弧は外してよい  
例:  $(\lambda x.(MN)) \equiv (\lambda x.MN) \neq ((\lambda x.M)N)$   
つまりラムダ抽象の本体は最長一致\*3する。
4.  $(M(\lambda x.N))$  は  $(M\lambda x.N)$  と略してもよい
5. 部分式でないラムダ式全体を囲む括弧は外してよい  
例:  $(M_1 M_2) \equiv M_1 M_2 \quad (\lambda x.M) \equiv \lambda x.M$

## 1.6 自由変数と束縛変数

ラムダ式の中の変数には 2 種類ある。

- **束縛変数 (bound variable)** :  $\lambda$  抽象の仮引数となっている変数のこと。変数名の名前を変えることができる。  
式  $(\lambda x.\lambda y'.xy')y$  において変数  $y'$  は  $\lambda y'$  によって束縛されている。
- **自由変数 (free variable)** : 束縛されていない変数のこと。外から見える変数であり、勝手に名前を変えることができない。  
式  $(\lambda x.\lambda y'.xy')y$  において変数  $y$  は自由である。

$\lambda x.(fx)$  というのは  $x$  が束縛変数なので、名前を変えることができ  $\lambda y.f(fy)$  とすることができる。一方で  $f$  は自由変数であるので  $\lambda x.g(gx)$  とすることはできない。同様に  $\lambda y.g(gy)$  とすることもできない。これを Ocaml で示す。

```
# let y = (fun a -> fun b -> a);; (適当に自由変数 y を設定する)
val y : 'a -> 'b -> 'a = <fun>
# let z = (fun a -> fun b -> b);; (適当に自由変数 z を設定する)
val z : 'a -> 'b -> 'b = <fun>
# let e1 = (fun x -> y);; (束縛変数 x を受け取って y を作用させる関数である)
val e1 : 'a -> 'b -> 'c -> 'b = <fun>
```

\*3 検索条件に複数の候補が一致する場合に、最も長いものを優先あるいは選択すること

```
# let e2 = (fun x -> z);;           (束縛変数 x を受け取って z を作用させる関数である)
val e2 : 'a -> 'b -> 'c -> 'c = <fun>
# e1 "xxx" "yyy" "zzz";;          (e1 の実行結果)
- : string = "yyy"
# e2 "xxx" "yyy" "zzz";;          (e2 の実行結果. e1 と異なる)
- : string = "zzz"
# let e3 = (fun w -> z);;          (束縛変数 w を受け取って z を作用させる関数である)
val e3 : 'a -> 'b -> 'c -> 'c = <fun>
# e3 "xxx" "yyy" "zzz";;          (e3 の実行結果. e2 と同じになる)
- : string = "zzz"
```

このように自由変数  $y$  から  $z$  に変えてしまうと実行結果が変わってしまうことがわかる。一方で束縛変数  $x$  から  $w$  に変えても実行結果が変わらないことがわかる。これは数学でも一般的に同じである。たとえば、 $f(x) = x + 1$  という関数  $f$  と  $g(y) = y + 1$  という関数  $g$  は束縛変数  $x, y$  が異なっているだけで等しいとみなされる。

ラムダ式  $M$  の自由変数の集合について、 $FV(M)$  と表記し、 $M$  の構造にしたがって再帰的に定義する。

$$\begin{aligned} FV(x) &\stackrel{\text{def}}{=} \{x\} \\ FV(\lambda x.M) &\stackrel{\text{def}}{=} FV(M) \setminus \{x\} \\ FV(MN) &\stackrel{\text{def}}{=} FV(M) \cup FV(N) \end{aligned} \quad (1.6)$$

自由変数が  $x$  しかない場合は集合は  $\{x\}$  である。 $x$  が束縛変数であるとき、ラムダ式  $M$  に含まれる自由変数は  $M$  に含まれる自由変数から  $x$  を抜いたものである。 $\lambda x.(\lambda y.xyz)$  であるならば、 $M = \lambda y.xyz$  となり、ここでの自由変数の集合は  $\{x, z\}$  であるが、 $\lambda x.(\lambda y.xyz)$  において  $x$  は束縛変数であるため、 $FV(\lambda x.M = \lambda x.(\lambda y.xyz)) = FV(M = \lambda y.xyz) \setminus \{x\} = \{z\}$  となる。ラムダ式  $M, N$  がある場合、ラムダ式  $MN$  に含まれる自由変数は  $M, N$  それぞれに含まれる自由変数の集合の和であることがわかる。ここでラムダ式  $M$  に自由変数が何も含まれていない場合、つまり  $FV(M) = \{\}$  のとき、 $M$  は閉じたラムダ式という。または結合子 (combinator) であるという。参考程度に束縛変数の定義を以下に載せる。ラムダ式  $M$  の束縛変数の集合について、 $BV(M)$  と表記し、 $M$  の構造にしたがって再帰的に定義する。

$$\begin{aligned} BV(x) &\stackrel{\text{def}}{=} \{\} \\ BV(\lambda x.M) &\stackrel{\text{def}}{=} BV(M) \cup \{x\} \\ BV(MN) &\stackrel{\text{def}}{=} BV(M) \cup BV(N) \end{aligned} \quad (1.7)$$

## 1.7 ラムダ式の簡約

### 1.7.1 ラムダ計算の small-step semantics

ラムダ計算は関数型をはじめとする大多数のプログラミング言語と異なり、abstraction 本体も評価する。これを strong reduction といい、abstraction の本体を評価しないのを weak reduction ということもある。ラムダ計算の中心は  $\beta$  簡約 ( $\beta$ -reduction) である。定義は以下である。

$$(\beta) \quad (\lambda x.M)N \longrightarrow M[x \mapsto N] \quad (1.8)$$

この意味は  $M$  の中の束縛変数  $x$  を  $N$  に置き換えるということである。

簡単な例として、 $(\lambda x.xx)b$  とすると束縛変数である  $x$  を  $b$  に置き換えるので  $bb$  となる。またこの  $\beta$  簡約ができるかどうか (簡約が無限に続かない) ことを推論する規則として以下の 3 つを挙げる。

$$\frac{M \longrightarrow M'}{MN \longrightarrow M'N} \quad \frac{M \longrightarrow M'}{LM \longrightarrow LM'} \quad \frac{M \longrightarrow M'}{\lambda x.M \longrightarrow \lambda x.M'} \quad (1.9)$$

この方法は様々な評価戦略がありうる。

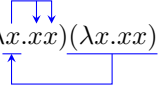


## 1.7.2 評価の行く末

詳しくは 1.12 にて扱う。評価戦略の行く末は以下のいずれかである。

- (1) いずれ簡約できなくなる  $\rightarrow$  正規形
- (2) 簡約が無限に続く (normal form)

例： $(\lambda x.x)y$  は正規形ではないが、正規形  $y$  をもつ。

$$\Omega \triangleq (\lambda x.xx)(\lambda x.xx) \text{ は正規形を持たない}$$


正規形となるか否かはいずれかである。

- (a) 評価戦略によらず (1)
- (b) 評価戦略によって (1) または (2)
- (c) 評価戦略によらず (2)

ここで、簡約が無限に続く  $\Omega = (\lambda x.xx)(\lambda x.xx)$  は

$$\Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \dots$$

となり、一方で  $(\lambda x.y)\Omega$  を考えると  $x$  を食べて  $\Omega$  を入れる  $\beta$  簡約を行えば

$$(\lambda x.y)\Omega \rightarrow y$$

となる。しかし、 $\Omega$  をずっと  $\beta$  簡約すると

$$(\lambda x.y)\Omega \rightarrow (\lambda x.y)\Omega \rightarrow (\lambda x.y)\Omega \rightarrow \dots$$

となり、簡約が無限に続く。このように評価戦略によっては簡約が終わる場合と終わらない場合が存在する。正規形を持つ場合、その評価戦略の仕方は 1 通りとは限らない。しかし必ず正規形の形は同じとなる。たとえば、 $(\lambda x.(\lambda y.xy)z)w$  においては以下ようになる。青は  $x$  から簡約した場合、赤は  $y$  から簡約した場合である。

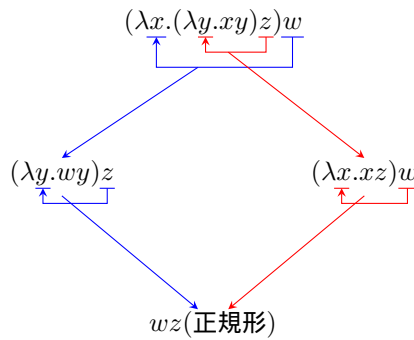


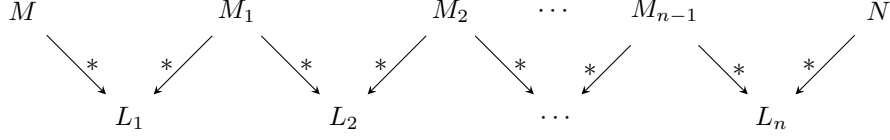
図 1.8  $(\lambda x.(\lambda y.xy)z)w$  の評価戦略

他にも、 $(\lambda x.ax)((\lambda y.by)c)$  についても、以下のように 2 通りで  $\beta$  簡約することができる。

$$\begin{aligned} (\lambda x.ax)((\lambda y.by)c) &\rightarrow a((\lambda y.by)c) \rightarrow a(bc) \\ (\lambda x.ax)((\lambda y.by)c) &\rightarrow (\lambda x.ax)(bc) \rightarrow a(bc) \end{aligned}$$

## 1.7.3 ラムダ式の等しさ

簡約規則を表す 2 項関係  $\rightarrow$  は同値関係ではない。二つのラムダ式が等しい ( $M = N$ ) とは「0 回異常の簡約 (reduction) とその逆操作 (expansion) によって  $M$  を  $N$  に変形できること」、つまり以下のような関係になればよい。ここで、 $\xrightarrow{*}$  は  $\rightarrow$  の反射推移閉包である。

図 1.9  $M = N$  の関係図

ゆえに  $=$  というのは  $\rightarrow$  の反射推移閉包  $(\rightarrow \cup \leftarrow)^*$  である。例えば、 $a((\lambda y.by)c) = (\lambda x.ax)(bc)$  である。これはどちらも簡約されて  $a(bc)$  となるためである。この関係  $(\rightarrow \cup \leftarrow)^*$  は  $\rightarrow$  を含む最小の同値関係 (equivalence relation) である。また下の関係が成り立つ。

$$M = M \quad \frac{M = N}{N = M} \quad \frac{L = M \quad M = N}{L = N} \quad (1.10)$$

ここで、 $(\rightarrow \cup \leftarrow)^*$  が  $\rightarrow$  を含む最小の同値関係であることを示す。

反射律は  $(\rightarrow \cup \leftarrow)^0 = (\equiv)$  より成立する。

推移律は任意の  $\lambda$  式  $A, B, C$  に対し、

$$\begin{aligned} A(\rightarrow \cup \leftarrow)^* B \wedge B(\rightarrow \cup \leftarrow)^* C &\iff \exists n, m \in \mathbb{N}. A(\rightarrow \cup \leftarrow)^n B \wedge B(\rightarrow \cup \leftarrow)^m C \\ &\iff \exists n, m \in \mathbb{N}. A(\rightarrow \cup \leftarrow)^{n+m} C \end{aligned}$$

となるので、成立する。

対称律は任意の  $\lambda$  式  $A, B$  に対し、 $A \leftarrow B \iff B \rightarrow A$  なので、

$$A(\rightarrow \cup \leftarrow) B \iff B(\rightarrow \cup \leftarrow) A$$

が成立することから成立する。

次に  $\rightarrow$  を含む最小の同値関係を  $R_m$  とし、 $(\rightarrow \cup \leftarrow)^* = R_m$  を示す。これは  $(\rightarrow \cup \leftarrow)^* \subseteq R_m$  を示せばよい。

そこで、 $n$  に関する帰納法により

$$\forall \in \mathbb{N}. (\rightarrow \cup \leftarrow)^n \in R_m$$

を示す。

まず、 $n = 0$  のとき

$$\begin{aligned} A(\rightarrow \cup \leftarrow)^0 B &\iff A \equiv B \\ &\iff AR_m B \quad (R_m \text{ の反射律}) \end{aligned}$$

より、 $(\rightarrow \cup \leftarrow)^0 \subseteq R_m$  が成立。

$n = 1$  のとき

$$\begin{aligned} A \rightarrow B &\iff AR_m B \\ A \leftarrow B &\iff B \rightarrow A \\ &\iff BR_m A \\ &\iff AR_m B \end{aligned}$$

より,  $(\rightarrow \cup \leftarrow) \subseteq R_m$  が成立.

ある  $n \in \mathbb{N}$  以下の全ての自然数で成り立つと仮定する. すると

$$\begin{aligned} A(\rightarrow \cup \leftarrow)^{n+1}B &\iff \exists C \in \Lambda. A(\rightarrow \cup \leftarrow)C \wedge C(\rightarrow \cup \leftarrow)^n B \\ &\iff \exists C \in \Lambda. AR_m C \wedge CR_m B \\ &\iff AR_m B \end{aligned}$$

となるので,  $n+1$  の場合でも成立することが分かる. よって

$$\forall n \in \mathbb{N}. (\rightarrow \cup \leftarrow)^n \in R_m$$

が成り立ち,

$$(\rightarrow \cup \leftarrow)^* = \bigcup_{n \in \mathbb{N}} (\rightarrow \cup \leftarrow)^n \subseteq R_m$$

なので,  $(\rightarrow \cup \leftarrow)^*$  は  $\rightarrow$  を含む最小の同値関係である.

$=$  は等価である関係としてきているが, ラムダ式の構成法に関する **合同関係** でもある. つまり以下が成り立つ.

$$\frac{M = M'}{MN = M'N} \quad \frac{M = M'}{LM = LM'} \quad \frac{M = M'}{\lambda x.M = \lambda x.M'} \quad (1.11)$$

#### 1.7.4 $\eta$ 簡約と $\eta$ 展開

$\eta$  簡約と  $\eta$  展開の定義式は以下ようになる.

$$(\eta) \quad (\lambda x.Mx) \longleftrightarrow M \text{ if } x \notin \text{FV}(M) \quad (1.12)$$

左から右の操作を  **$\eta$  簡約** (eta-reduction), 右から左の操作を  **$\eta$  展開** (eta-expansion), 両方の操作を合わせて  **$\eta$  変換** (eta-conversion) という. たとえば

$$(\lambda x.(yz))x \longleftrightarrow yz$$

という風になる. この  $\eta$  変換は関数としての **外延的な等しさ** を表現していて「実行」ではなく, 一種の変形作業である.  $\eta$  簡約の有無は, ラムダ計算の本質的な部分には影響しない.

### 1.8 $\alpha$ 変換

変数を代入するときのルールを以下に示す. このとき代入を表す  $[y \mapsto L]$  は **メタ記法** であり, つまりラムダ計算について議論するための記法であって, ラムダ計算自身の記法ではない. ここで,  $x, y$  は変数であり,  $M, N, L$  はラムダ式である.

$$\begin{aligned} x[y \mapsto L] &\stackrel{\text{def}}{=} \begin{cases} L & \text{if } x \equiv y \\ x & \text{if } x \not\equiv y \end{cases} \\ (MN)[y \mapsto L] &\stackrel{\text{def}}{=} (M[y \mapsto L])(N[y \mapsto L]) \\ (\lambda x.M)[y \mapsto L] &\stackrel{\text{def}}{=} \begin{cases} \lambda x.M & \text{if } x \equiv y \\ \lambda x.(M[y \mapsto L]) & \text{if } x \not\equiv y \wedge x \notin \text{FV}(L) \\ \lambda z.(M[x \mapsto z])[y \mapsto L] & \text{if } x \not\equiv y \wedge x \in \text{FV}(L) \wedge z \notin \text{FV}(M) \cup \text{FV}(L) \end{cases} \end{aligned} \quad (1.13)$$

これを整理すると次のようになる.

- ①  $(\lambda x.x)L = L$
- ②  $(\lambda x.y)L = y$
- ③  $(\lambda x.(MN))L = ((\lambda x.M)L)((\lambda x.N)L)$
- ④  $(\lambda x.(\lambda x.M))L = \lambda x.M$

$$(5) (\lambda y.(\lambda x.M))L = \lambda x.((\lambda y.M)L)$$

ここで変数の衝突が起こらないように  $x \notin \text{FV}(L)$  かそもそも  $y \notin \text{FV}(M)$  である.

$$(6) (\lambda y.(\lambda x.M))L = (\lambda y.(\lambda z.(\lambda x.M)z))L$$

$y \in \text{FV}(M)$  かつ  $x \in \text{FV}(L)$  のとき,  $(\lambda x.M)$  での束縛変数  $x$  を  $z$  に書き換え, 変数の衝突 (変数補足, variable capture) を回避している. ただし,  $z$  は  $M$  にも  $L$  にも現れない変数.

たとえば,  $M = xyw$ ,  $L = xw$  とすると以下ようになる.

$$\begin{aligned} (\lambda y.(\lambda x.xyw))(xw) &= (\lambda y.(\lambda z.zyw))(xw) \\ &= \lambda z.zxww \end{aligned}$$

先の (6) のような場合の, 変数捕捉を避けるための変数変換を  $\alpha$  変換という. 定義は以下のとおりである.

$$(\alpha) \quad \lambda x.M \longrightarrow \lambda y.M[x \mapsto y] \quad \text{if } y \notin \text{FV}(M) \quad (1.14)$$

この変換に関しては計算の観点から本質的とは言えないので,  $\alpha$  同値つまり  $\alpha$  変換によって同一になれるラムダ式は同一視する (“ $\equiv$ ” の関係にあるとみなす).

## 1.9 ラムダ式の記述能力

### 1.9.1 ブール値

ラムダ式はさまざまなデータや制御構造を表現 (エンコード) することができる. 例えば必須の制御機能である条件判断や, 再帰呼び出し, データ及びデータ構造であるブール値や自然数, 順序対などが表現可能である. ブール値と条件判断は以下の定義で表現する.

$$\overline{\text{true}} \stackrel{\text{def}}{=} \lambda xy.x \quad (1.15)$$

$$\overline{\text{false}} \stackrel{\text{def}}{=} \lambda xy.y \quad (1.16)$$

$$\overline{\text{if}} \stackrel{\text{def}}{=} \lambda pxy.pxy \quad (1.17)$$

$$\overline{\text{not}} \stackrel{\text{def}}{=} \lambda x.x \overline{\text{false}} \overline{\text{true}} \quad (1.18)$$

if  $p$  then  $x$  else  $y$  という構文を考える. これに if true then  $x'$  else  $y'$  とすると

$$\begin{aligned} \overline{\text{if true}} x'y' &= (\lambda pxy.pxy) \overline{\text{true}} x'y' \\ &\rightarrow \overline{\text{true}} x'y' \\ &\rightarrow (\lambda xy.x) x'y' \\ &\rightarrow x' \end{aligned}$$

同様にして false の場合は

$$\begin{aligned} \overline{\text{if false}} x'y' &= (\lambda pxy.pxy) \overline{\text{false}} x'y' \\ &\rightarrow \overline{\text{false}} x'y' \\ &\rightarrow (\lambda xy.y) x'y' \\ &\rightarrow y' \end{aligned}$$

$\overline{\text{not}}$  に関して, if  $x$  then false else true と考えることができれば,

$$\begin{aligned} \overline{\text{not}} &= \lambda x.(\overline{\text{if}} x \overline{\text{false}} \overline{\text{true}}) \\ &\rightarrow \lambda x.((\lambda pxy.pxy)x \overline{\text{false}} \overline{\text{true}}) \\ &\rightarrow \lambda x.x \overline{\text{false}} \overline{\text{true}} \end{aligned}$$

$\overline{\text{and}}$  について考えてみる. これは変数  $x, y$  に対し if  $x$  then (if  $y$  then true else false) else false と考えることができれば,

$$\begin{aligned} \overline{\text{and}} &= \lambda xy.\overline{\text{if}} x (\overline{\text{if}} y \overline{\text{true}} \overline{\text{false}}) \overline{\text{false}} \\ &= \lambda xy.xy \overline{\text{true}} \overline{\text{false}} \overline{\text{false}} \\ &= \lambda xy.xy \overline{\text{false}} \end{aligned}$$

$\overline{\text{or}}$  についても考えてみる。これは変数  $x, y$  に対し  $\text{if } x \text{ then true else } y$  と考えることができるので

$$\begin{aligned}\overline{\text{or}} &= \lambda xy. \overline{\text{if } x \text{ true}} y \\ &= \lambda xy. x \overline{\text{true}} y\end{aligned}$$

### 1.9.2 順序対 (tuple)(=直積型の要素)

順序対は基本データを組合せて複雑なデータを表現するのに使用する。例えば複素数は、実部と虚部を表す実数の組として表現可能であり、整数は自然数と符号の組として表現することが可能である。 $\overline{\text{pair}}$  を考えてみる。これは組の要素にアクセスする関数  $f$  を受け取り、 $x, y$  に適用する関数として表現する。この定義式は以下になる。

$$\overline{\text{pair}} \stackrel{\text{def}}{=} \lambda xyf. fxy = \lambda xy. \lambda f. fxy \quad (1.19)$$

この組に対して最初の要素を取り出す、最後の要素を取り出すラムダ式  $\overline{\text{head}}$ ,  $\overline{\text{tail}}$  は以下のように定義できる

$$\overline{\text{head}} \stackrel{\text{def}}{=} \lambda xy. x \quad (\equiv \overline{\text{true}}) \quad (1.20)$$

$$\overline{\text{tail}} \stackrel{\text{def}}{=} \lambda xy. y \quad (\equiv \overline{\text{false}}) \quad (1.21)$$

これは単純に二つの変数を受け取って最初の変数を返すものが  $\overline{\text{head}}$  であり、二つめの変数を返すものが  $\overline{\text{tail}}$  であるというだけである。

これである場合、組での最初の要素を取り出すということとはできない。あくまでも二つの変数の内最初を取り出す、または最後を取り出すということだけである。これを組でも適用させるには後置演算子<sup>\*4</sup>として利用する方法である。つまり  $M$  と  $N$  の順序対を作った場合、最初の要素である  $M$  を取り出すには

$$\begin{aligned}\overline{\text{pair}} \ MN \ \overline{\text{head}} &= (\lambda xyf. fxy)MN \ \overline{\text{head}} \\ &\rightarrow (\lambda yf. fMy)N \ \overline{\text{head}} \\ &\rightarrow (\lambda f. fMN) \ \overline{\text{head}} \\ &\rightarrow \overline{\text{head}} \ MN \\ &\rightarrow \overline{\text{true}} \ MN \\ &\rightarrow (\lambda xy. x)MN \\ &\rightarrow M\end{aligned}$$

最初に演算子を宣言する方法は組  $\overline{\text{pair}}$  の最初の要素を取り出すものを  $\overline{\text{fst}}$  とし、最後の要素を取り出すものを  $\overline{\text{snd}}$  とし、以下に定義を載せる。

$$\overline{\text{fst}} \stackrel{\text{def}}{=} \lambda p. p \ \overline{\text{true}} \quad (1.22)$$

$$\overline{\text{snd}} \stackrel{\text{def}}{=} \lambda p. p \ \overline{\text{false}} \quad (1.23)$$

$\overline{\text{fst}}$  の使い方は以下になる。

$$\begin{aligned}\overline{\text{fst}} (\overline{\text{pair}} \ MN) &= \overline{\text{fst}} (\lambda f. fMN) \\ &\rightarrow (\lambda f. fMN) \ \overline{\text{true}} \\ &\rightarrow \overline{\text{true}} \ MN \\ &\rightarrow M\end{aligned}$$

このようにして順序対をラムダ式で表現することが可能となった。順序対ができれば直和型の要素も表現することが可能となる。

<sup>\*4</sup> 変数を宣言した後に演算子を置くこと。プログラミングにおける  $i++$  にあたる。

## 1.9.3 自然数とその算術

自然数をラムダ式においては **Church numerals**(チャーチ数) と呼ばれるエンコード法で表現する。 $x \leftrightarrow \text{zero}$  とし,  $f \leftrightarrow \text{successor}$  とする。これは Prolog の自然数表現でも用いられる。

$$\begin{aligned}\bar{0} &\stackrel{\text{def}}{=} \lambda f x. x \\ \bar{1} &\stackrel{\text{def}}{=} \lambda f x. f x \\ \bar{2} &\stackrel{\text{def}}{=} \lambda f x. f(f x) \\ \bar{3} &\stackrel{\text{def}}{=} \lambda f x. f(f(f x))\end{aligned}$$

このように  $f$  を自然数  $n$  に対して  $n$  回  $x$  に適用させていると考えることができる。

またこの自然数を元に  $\overline{\text{add}}$  と  $\overline{\text{mul}}$ ,  $\overline{\text{exp}}$  は次のように表現することができる。

$$\overline{\text{add}} \stackrel{\text{def}}{=} \lambda m n. \lambda f x. n f (m f x) \quad (1.24)$$

$$\overline{\text{mul}} \stackrel{\text{def}}{=} \lambda m n. \lambda f x. n (m f) x \quad (= \lambda m n. \lambda f. n (m f)) \quad (1.25)$$

$$\overline{\text{exp}} \stackrel{\text{def}}{=} \lambda m n. n m \quad (1.26)$$

以下, これを証明する。まず  $f(f(f(\dots x)))$  と  $f$  が  $n$  回適用された場合は  $f^n$  と表すことにする。すると  $f^n$  は  $n f$  と表現することができる。ここで  $n$  はチャーチ数であることから  $n$  で  $f(f(f(\dots x)))$  を表現することに注意する。

$$\begin{array}{lll}\overline{\text{add}} = \lambda m n. \ulcorner m + n \urcorner & \overline{\text{mul}} = \lambda m n. \ulcorner m \times n \urcorner & \overline{\text{exp}} = \lambda m n. \ulcorner m^n \urcorner \\ = \lambda m n. \lambda f x. f^{m+n} x & = \lambda m n. \lambda f x. f^{m \times n} x & = \lambda m n. \lambda f x. f^{m^n} x \\ = \lambda m n. \lambda f x. f^m (f^n x) & = \lambda m n. \lambda f x. (f^m)^n x & = \lambda m n. \lambda f x. m^n f x \\ = \lambda m n. \lambda f x. m f (n f) x & = \lambda m n. \lambda f x. (m f)^n x & = \lambda m n. \lambda f x. n m f x \\ & = \lambda m n. \lambda f x. n (m f) x & \rightarrow \lambda m n. n m\end{array}$$

これを用いれば  $0^0$  を計算することができる。

$$\begin{aligned}\overline{\text{exp}} \bar{0} \bar{0} &= (\lambda m n. n m) (\lambda f x. x) (\lambda f x. x) \\ &\rightarrow (\lambda n. n (\lambda f x. x)) (\lambda f x. x) \\ &\rightarrow (\lambda f x. x) (\lambda f x. x) \\ &\rightarrow \lambda x. x\end{aligned}$$

となる。最後の行は  $f$  を食べて  $(\lambda f x. x)$  をうけるとが  $\lambda f x. x$  より  $\lambda$  式に  $f$  が存在しないので,  $\lambda x. x$  となる。ここで,  $\lambda x. x$  というのは  $x$  を食べて  $x$  を返す関数, つまり 1 と考えることができる。これより  $\lambda$  式においては  $0^0 = 1$  とみなすことが可能である。

他の算術関数を挙げる。

$$\overline{\text{iszero}} \stackrel{\text{def}}{=} \lambda n. n (\lambda x. \overline{\text{false}}) \overline{\text{true}} \quad (1.27)$$

$$\overline{\text{pre}} \stackrel{\text{def}}{=} \lambda n. \lambda f x. \overline{\text{snd}} (n (\overline{\text{prefn}} f) (\overline{\text{pair}} x x)) \quad (1.28)$$

$$\overline{\text{prefn}} \stackrel{\text{def}}{=} \lambda f p. \overline{\text{pair}} (f (\overline{\text{fst}} p)) (\overline{\text{fst}} p) \quad (1.29)$$

## 1.9.4 再帰関数

今までの例からラムダ式が**原始再帰関数** (直感的には for ループでかける関数) の表現力を持っていることは明らかである。一方で**再帰関数** (直感的には while ループがないとかけない関数) は表現できるかどうかということである。ラムダ計算の特徴は無名関数を表現できることであるが, 関数に名前をつけないで再帰を表現するには**不動点結合子**を使う。

## 1.10 不動点結合子

Church-Turing thesis(チャーチ・チューニングのテーゼ) から  
(何らかのアルゴリズムで) 計算可能ということをラムダ式で定義可能と定義し, これは Turing マシンで計算可能であることと等価であり,  $\mu$  再帰関数 ( $\mu$ -recursive functions, 帰納的関数) である.

### 1.10.1 不動点定理

不動点とは一般に写像によって自分自身に写される点のことである. つまり  $x$  が  $f$  の不動点とは  $f(x) = x$  である. 実数関数の場合の例を以下に挙げる.

$$\begin{cases} \lambda x.x^2 & - \text{複数の不動点 (0 と -1)} \\ \lambda x.x + 1 & - \text{不動点なし} \\ \lambda x.x & - \text{無数の不動点} \end{cases}$$

ラムダ計算において不動点の定義は以下の不動点定理によるものである.

$$\forall F \in \Lambda \exists X \in \Lambda (FX = X) \quad (1.30)$$

このとき  $X$  を  $F$  の不動点という. 以下証明する.

$X$  を次のようにおく.

$$X \triangleq (\lambda x.F(xx))(\lambda x.F(xx))$$

こうおくことによって, 矢印のように代入されるので

$$X = FX$$

となる. ゆえに  $X$  は  $F$  の不動点となる.

これと同様の議論して  $Y$  を **Curry の不動点結合子** というものを定義する. 定義式は以下ようになる.

$$Y \triangleq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \quad (1.31)$$

これから,

$$YF = F(YF)$$

つまり,  $Y$  は任意の関数  $F$  を引数にとり, その  $F$  の不動点を与える関数である.

### 1.10.2 再帰関数の定義

再帰関数の例として

`fact(n) = if n=0 then 1 else n*fact(n-1)`

というものを考える. 関数  $g$  について書いてみると,  $g = \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * g(n-1)$  と直せるのでこの  $f$  を食べる関数を  $\overline{\text{factgen}}$  とすれば

$$\overline{\text{factgen}} = \lambda g. \lambda n. \overline{\text{if}} (\overline{\text{iszero}} n) \overline{1} (\overline{\text{mul}} n (g(\overline{\text{pre}} n))) \quad (1.32)$$

とかける. ここで求めたい再帰関数  $\overline{\text{fact}}$  に関しては,

$$g = \overline{\text{factgen}} g$$

という関係式が成り立つこと, つまり  $\overline{\text{factgen}}$  の不動点が  $\overline{\text{fact}}$  であることが言えるので, Curry の不動点結合子  $Y$  を用いれば求めることができ,

$$\begin{aligned} \overline{\text{fact}} &= Y \overline{\text{factgen}} \\ &\triangleq Y(\lambda f. \lambda n. \overline{\text{if}} (\overline{\text{iszero}} n) \overline{1} (\overline{\text{mul}} n (f(\overline{\text{pre}} n)))) \end{aligned} \quad (1.33)$$

## 1.10.3 再帰関数の意味

この分野はスライドに書いてあることを軽く載せる.

階乗関数を次のように定義しておく.

$$\mathbf{F} = \lambda g. \lambda n. \overline{\text{if}} (\overline{\text{iszero}} n) \overline{1} (\overline{\text{mul}} n (g(\overline{\text{pre}} n))) \quad (1.34)$$

$$\Phi = \lambda n. \perp \quad (1.35)$$

$\perp$  は undefined と解釈する. ラムダ式  $M$  の, 自然数上の部分関数としての解釈を  $\llbracket M \rrbracket_{N \rightarrow N}$  とかく. これをから  $\llbracket \Phi \rrbracket$ ,  $\llbracket \mathbf{F}(\Phi) \rrbracket$ ,  $\llbracket \mathbf{F}^2(\Phi) \rrbracket$ ,  $\llbracket \mathbf{F}^3(\Phi) \rrbracket$ , ... を考えると以下のように階乗関数の近似列になる.

$$\begin{aligned} \llbracket \Phi \rrbracket &= \{ \} \\ \llbracket \mathbf{F}(\Phi) \rrbracket &= \{ \langle 0, 1 \rangle \} \\ \llbracket \mathbf{F}^2(\Phi) \rrbracket &= \{ \langle 0, 1 \rangle, \langle 1, 1 \rangle \} \\ \llbracket \mathbf{F}^3(\Phi) \rrbracket &= \{ \langle 0, 1 \rangle, \langle 1, 1 \rangle, \langle 2, 2 \rangle \} \end{aligned}$$

これから結合子について以下が成り立つことがいえる.

1.  $\llbracket M_1 \rrbracket \subseteq \llbracket M_2 \rrbracket \Rightarrow \llbracket \mathbf{F}(M_1) \rrbracket \subseteq \llbracket \mathbf{F}(M_2) \rrbracket$
2.  $\llbracket \mathbf{F}^n(\Phi) \rrbracket \subseteq \llbracket \mathbf{F}^{n+1}(\Phi) \rrbracket$
3.  $\llbracket \mathbf{YF} \rrbracket = \llbracket \bigcup \mathbf{F}^n(\Phi) \rrbracket = !$  (階乗関数)
4.  $\llbracket \mathbf{YF} \rrbracket_{N \rightarrow N}$  は  $\llbracket \mathbf{F} \rrbracket_{(N \rightarrow N) \rightarrow (N \rightarrow N)}$  の最小不動点である

## 1.10.4 様々な不動点結合子

まず Curry の不動点結合子は以下のように定義されていた.

$$Y \triangleq \lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$$

これから無限データ構造を定義することが可能である.

$$\overline{\text{zeroes}} \triangleq Y(\lambda g. \overline{\text{pair}} \overline{0} g) \quad (1.36)$$

これを図示すると以下に様にかくことができる.

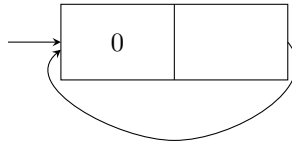


図 1.10 無限データ構造

次に Turning の不動点結合子は以下のように定義されている.

$$\Theta \triangleq (\lambda xy. y(xy))(\lambda xy. y(xy)) \quad (1.37)$$

この Turning の不動点結合子について,  $\Theta F \xrightarrow{*} F(\Theta F)$  であることを確かめる.

$$\begin{aligned} \Theta F &= (\lambda xy. y(xy))(\lambda xy. y(xy)) F \\ &= (\lambda xy. y(xy))(\lambda zw. w(zzw)) F \\ &= (\lambda y. y((\lambda zw. w(zzw))(\lambda zw. w(zzw)) y)) F \\ &= F((\lambda zw. w(zzw))(\lambda zw. w(zzw)) F) \\ &= F(\Theta F) \end{aligned}$$



## 1.10.5 不動点結合子と Russell のパラドックス

ラムダ計算は当初, 数学の体系化 (Hilbert のプログラム) のための記法として着想された. 論理体系も以下のように定義しようとしていた

集合		λ 式
$x \in S$	$\iff$	$Sx$
$\{x P\}$	$\iff$	$\lambda x.P$

しかし Russel のパラドックスによって, 論理体系としては使えないことが判明. ある命題とその否定が等価であることが導けてしまう.

以下のように集合を定義する.

$$R \triangleq \{x \mid x \notin x\} \iff R \triangleq \lambda x. \overline{\text{not}}(xx)$$

これに対して  $R \in R$  というのは

$$R \in R \iff RR = (\lambda x. \overline{\text{not}}(xx))(\lambda x. \overline{\text{not}}(xx)) = \overline{\text{not}}(RR)$$

となる. だが, 計算の体系としては今なお健在である.

## 1.11 Church-Rosser 定理

Church-Rosser 定理とはもし  $M_1 = M_2$  という λ 式が存在する時 ( $M_1$  と  $M_2$  が  $(\rightarrow \cup \leftarrow)^*$  の関係), 必ず  $M_1 \xrightarrow{*} N$  かつ  $M_2 \xrightarrow{*} N$  となる λ 式  $N$  が存在するというものである. この性質は, “計算結果は計算手順によらない” ことを意味している. これを図式化すると以下のように書くことができる.

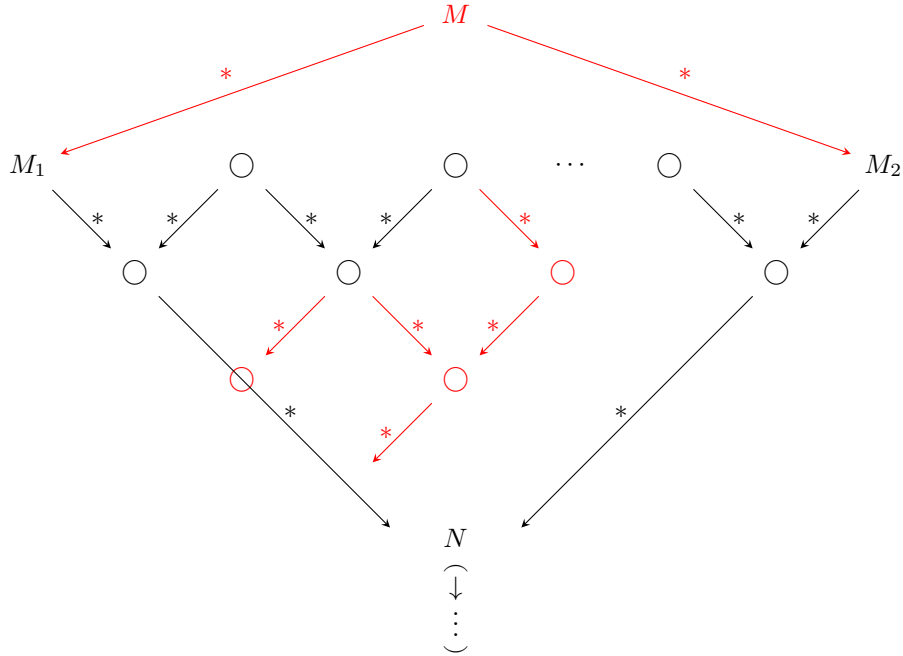


図 1.11 Church-Rosser 定理

この図 1.11 を大きくみると diamond のような形をしていることがわかる. つまり,  $M \xrightarrow{*} M_1$  と  $M \xrightarrow{*} M_2$  を満たし, かつ  $M_1 \xrightarrow{*} N$  と  $M_2 \xrightarrow{*} N$  を満たすときは構造上 diamond のような形をしている. この構造は **Diamond Property** とよばれる.

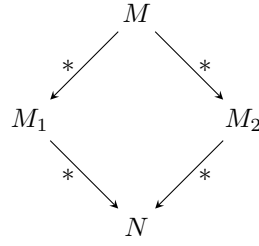


図 1.12 Diamond Property

一方で,  $M \rightarrow M_1$ ,  $M \rightarrow M_2$ ,  $M_1 \rightarrow N$ ,  $M_2 \rightarrow N$  という関係は必ずしも成立しない.

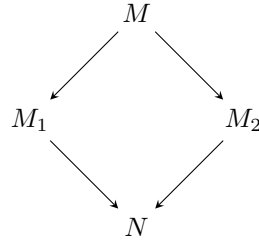


図 1.13 Diamond property の例外

図 1.13 の例外として,  $(\lambda x.xx)((\lambda x.x)a)$  というものがある. グラフは以下ようになる.

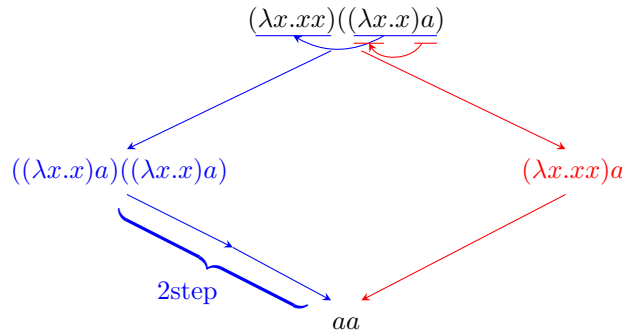
図 1.14  $(\lambda x.xx)((\lambda x.x)a)$  の簡約グラフ

図 1.13 の Diamond property が成立する  $\lambda$  式である場合, 図 1.11 は diamond property を繰り返し施すことによって成立することが自明に従う. 一方でこの diamond property を必ずしも持たないため Church-Rosser 定理は自明ではない. この定理から導けるもの系を以下に載せる.

- $M = N$  かつ  $N \not\rightarrow^*$  (正規形) ならば,  $M \rightarrow^* N$ <sup>\*5</sup>
- $M = N$  かつ  $M \not\rightarrow$  かつ  $N \not\rightarrow$  ならば  $M \equiv N$  (up to  $\alpha$ )
- $M \not\rightarrow$  かつ  $N \not\rightarrow$  かつ  $M \neq N$  ならば  $M \neq N$

たとえば,  $\lambda xy.x(= \text{true})$  と  $\lambda xy.y(= \text{false})$

これからすべての  $\lambda$  式が等価になってしまうことはない. これを **ラムダ計算は等式理論として無矛盾である** という.

この系から以下のことがいえる.

<sup>\*5</sup> 今までは  $\rightarrow^*$  と表現していた. 正しくは  $\rightarrow^*$  のほうであるが, 図との対応をつけるために  $\rightarrow^*$  としていた.

- 注 1:  $\eta$  簡約を認めると  $\lambda y.xy = x$  だが, 認めなければ,  $\lambda y.xy \neq x$  である. よって, このことから  $\eta$  簡約は冗長なルールではないことがいえる.
- 注 2: Church-Rosser 定理は, 等しい  $\lambda$  式には簡約の合流点があることを保証しているだけで, 簡約の停止性を保証しているわけではない.
- 注 3: Church-Rosser 定理から, Diamond Property とよばれる性質が導かれ, これを Church-Rosser ということもある.

## 1.12 標準化定理とリダクション戦略

リダクション (簡約) の代表戦略として **Normal order** というものがある. これは  $\beta$  基  $((\lambda x.M)N)$  の形のもので,  $\lambda$  が一番左のものを簡約する戦略である. 最外最左簡約ともよばれ,  $\lambda$  計算において簡約は有限ステップで正規系に到達せず, 計算が終了しない場合があるが, normal order reduction に関しては正規形に到達する簡約列がひとつあるなら, normal order reduction でも必ず正規形に到達するという性質がある. この性質を**標準化定理**という.

$\lambda$  式を記号列で考えた場合は leftmost という.  $MN$  の形の式で,  $M$  が  $(\lambda x.M')$  の形でなければまず  $M$  を簡約する方法である.

$\lambda$  式を抽象構文木で考えた場合は leftmost outermost とよばれる.  $(\lambda x.M)N$  の形の式をまず  $M[x \mapsto N]$  の形にすることを outermost という.

Normal order と似ている戦略として call by name とよばれるものがある. これは Algol, Miranda, Haskell といった言語では, 引数の計算をする前に関数呼び出しが発生する. この引数が関数本体内で必要になったときにのみ, その値が計算される方法のことを call by name (名前呼び) という.

他のリダクション戦略として **Applicative order** というものがある. これは  $\beta$  基を中に含まない  $\beta$  基 (の中の最左のもの) を簡約する方法である. この戦略は innermost や leftmost innermost とよばれる.

Applicative order と似ている戦略として call by value とよばれるものがある. 多くのプログラミング言語では関数/手続呼び出しは, 引数の計算が終わってから行われる. このような呼び出し方法を call by value (値呼び) とよぶ.

$\Omega \triangleq (\lambda x.xx)(\lambda x.xx)$  について,  $(\lambda x.y)\Omega$  を normal order と applicative order での戦略での簡約を見てみる. normal order 戦略からやると, *lambdax* から  $\beta$  簡約を行うので,

$$(\lambda x.y)\Omega \rightarrow y$$

となる. 一方で, applicative order 戦略からやると  $\Omega$  を簡約するが, この簡約は停止しないので

$$(\lambda x.y)\Omega \rightarrow (\lambda x.y)\Omega \rightarrow (\lambda x.y)\Omega \rightarrow \dots$$

となってしまう.

このため, normal order 戦略で簡約すると有限ステップで停止するので (正規形が存在するならば), normal order を用いたほうが良いと考えられるが, 多くの言語では applicative order と同じような call by value を採用している. この理由として call by value は call by name に比べて実装が容易であるためである. 関数閉包という方法で call by name と同じことができるが実際はほとんど使われていない. その理由として以下の  $\lambda$  式 **sgr** を挙げる.

$$\overline{\text{sgr}} \triangleq \lambda n. \overline{\text{mul}} \, nn \quad (1.38)$$

これから  $\overline{\text{sgr}} \, (\overline{\text{add}} \, 2 \, 3)$  を考えてみる. call by value の場合は,  $\overline{\text{add}} \, 2 \, 3$  を計算してから  $\overline{\text{sgr}}$  を適用させる. ゆえに引数を 1 回評価すればそれ以上使われることはない. 一方で call by name で行くと以下のようなになる.

$$\overline{\text{sgr}} \, (\overline{\text{add}} \, 2 \, 3) \rightarrow \overline{\text{mul}} \, (\overline{\text{add}} \, 2 \, 3) \, (\overline{\text{add}} \, 2 \, 3)$$

これによって引数のコピーが 2 個作られるので、関数実行回数が増えてしまう。これはプログラムとして計算量が増えてしまう原因となる。そのため Normal order の考えが必ずしもよいとはいえない。この Normal order の考え、つまり call by name の計算量が増えるのを防ぐ方法として遅延評価 (lazy evaluation) というものがある。これはプログラミング言語においては call by need とよばれる (Haskell などの言語で使われる)。使わない引数を評価しないことや、一度使った引数を保存する、つまり評価結果を共有するという方法である。このことによって計算量を減らすことができる。引数評価についてまとめると以下ようになる。

- call by value : 引数をちょうど 1 回評価
- call by name : 引数を使った回数評価
- call by need : 引数を高々 1 回評価

## 2 型体系 (type systems)

### 2.1 型体系の目的

機械語 (に近い) プログラムの中では、同一のビット列表現が異なる意味で用いられることがある。C/C++, Java などでは、integer の 0 と boolean の false は同一の意味で用いられる。また機械語においては命令列とデータは同じ 0 と 1 で表現される (これをはじめの何文字化を読み取ることで命令列とデータを区別している)。このことはラムダ計算でも同様である。たとえば、以下のものは同じ  $\lambda$  式である。

- $\overline{0} \stackrel{\text{def}}{=} \lambda f x. x$
- $\overline{\text{false}} \stackrel{\text{def}}{=} \lambda x y. y$
- $\overline{\text{tail}} \stackrel{\text{def}}{=} \lambda x y. y \quad (\equiv \overline{\text{false}})$

プログラミングの観点からは、ビット列や  $\lambda$  式自体よりも、それをどういう意味で使っているかが重要である。各変数のとりうる値や各関数の役割 (引数と戻り値) を構文的に類別 (classify) し、さらにその類別をプログラム実行がきちんと守ることを保証されるべきである。もし型の概念がない場合は次の計算が可能となってしまう。

$$\begin{aligned}
 \text{false} + 2 &= \overline{\text{add false 2}} \\
 &= (\lambda mn. \lambda f x. n f (m f x)) (\lambda x y. y) (\lambda f x. f (f x)) \\
 &\rightarrow^* \lambda f x. (\lambda f x. f (f x)) (f ((\lambda z y. y) f x)) \\
 &\rightarrow^* \lambda f x. (\lambda f x. f (f x)) (f x) \\
 &\rightarrow^* \lambda f x. f (f x) \\
 &= \overline{2}
 \end{aligned}$$

### 2.2 型体系の意義

型体系の意義は以下の 5 つである。

1. プログラミングやプログラム理解の助けになる
2. プログラムの (ある種の) 誤りの静的発見  
ゼロ除算, 無限ループ, 範囲外の配列添字, 等は大多数の言語では動的検出または不検出である。
3. プログラムの基本的な性質の静的保証
4. プログラム最適化のための基本情報の提供
5. モジュラープログラミングの促進  
(データ抽象化, インタフェース明確化)

この内, 1 と 5 は型の明示によって生まれる意義であり, 2,3,4 は型を明示しなくても可能な場合がある. 型の明示がなくても有益な情報が得られる例として以下の場合がある.

```
+:int × int    int
+: String × String  String
```

$x:int, y:int$  から  $x+y:int$  がわかる.

$String \times String$  というのは  $String$  変数  $+String$  変数とみなすことができ  $+$  の多重定義<sup>\*6</sup>となる. 次に

```
a := (if b then c else d)
```

というのを考えると,  $c$  と  $d$  という型は同じでないといけない. つまりこのことからどちらも同じ型であることが推定できる.

## 6. Curry-Howard 対応

一般的には Curry-Howard 同型対応<sup>\*7</sup>と呼ばれる. 簡単に言えば, 数学における証明の構造と, コンピュータ・プログラムの構造がぴったり一致するということである. つまり

型            ↔    論理式  
プログラム   ↔    証明

たとえば直積  $A \times B$  というのは論理積  $A \wedge B$  と対応していて, 関数  $A \rightarrow B$  というのは含意  $A \Rightarrow B$  と対応している. つまり,  $A \times B \rightarrow C$  というのは  $A \wedge B \Rightarrow C$  と対応している. Curry 化<sup>\*8</sup>に関して,  $A \times B \rightarrow C$  というのは  $A \rightarrow B \rightarrow C$  となる. これは  $A \wedge B \Rightarrow C$  が  $A \Rightarrow B \Rightarrow C$  となることと対応する ( $A$  をもらって  $B$  をもらって  $C$  をもらうことは  $A$  ならば  $B$  でありそれであるならば  $C$  であるに対応づいていると考えられる).

## 2.3 型とプログラムの安全性

安全なプログラミング言語は **a language that protects its own abstractions** と英語で呼ぶ. abstraction とは機械語レベルでは存在しないが, 高級言語 (high-level language) レベルに存在する概念のことである. 例えば, ポインタ, 参照 (reference), 配列と添字, 型である. abstraction を日本語で言えば抽象化である. 機械語レベルでは数字でしか認識しないので抽象化することはできない. 一方でポインタというものは値が格納されている場所を抽象化して用いている. つまりアドレスをもつ変数ではなく, 変数のもつアドレスを抽象化してポインタは利用している. したがって, abstraction には安全性の観点から保護が必要である. たとえばポインタや参照 (reference) の場合, そのポインタが指し示す場所が存在しているのか, 参照する先の値がきちんと用意されているのか保証されていることを確認しなければならない. 配列と添字に関しては配列のサイズ以上の添字を指定することで配列の範囲外を参照することになっていないか? ということを確認してそのようなことが起きないように保証されていないといけない. 型に関して, 足し算などの演算において同じ型同士で計算されているかどうか保証されていないといけない. また値を代入, 引数に入れる型が予想される通りの型であるか保証されていないといけない. 型が矛盾なく保証されていれば**型安全 (type safe)** であるという.

<sup>\*6</sup> 同じ演算子を他の型でも定義して用いること.  $1+2$  と “string”+“name” などにおいて  $+$  は多重定義となる.

<sup>\*7</sup> 同型対応とは構造が一致することである. 構造とはいくつかのモノたちと, それらの間の関係からなる骨格のことである. つまり同型対応とは中身が異なっても, 骨格が同じことを指す

<sup>\*8</sup> 複数の引数を取る関数を, 引数が「もとの関数の最初の引数」で戻り値が「もとの関数の残りの引数を取り結果を返す関数」であるような関数にすること (あるいはその関数のこと)

誤りを検出する方法と安全性を以下に示す。

表 2.1 誤りの検出方法と安全性

	利点	欠点
静的 (コンパイル時) 検査	安全性, 実行時 overhead の回避	保守的
動的 (実行時) 検査	安全性	実行時 overhead
無検査 (ex.C 言語の配列添字)		危険

コンパイラは“決定可能”な性質しか保証できない。配列添えの参照が参照外であるかどうかはコンパイラ事典では判別できない。添字参照に変数を用いている場合は実行時にしかその変数の値を求められないので、決定可能ではない。型安全性における各プログラミング言語でまとめると次のようになる。

表 2.2 型プログラムの安全性：具体例

アセンブラ	untyped, unsafe
Lisp, JavaScript	untyped(または dynamically typed), safe(pointer safety, etc.)
C	(statically) typed, unsafe(cast, pointer arithmetic, etc.) (C を safe にする研究もある)
Java	(statically) typed, safe (after 1997)
ML, Scala	(statically) typed, safe

単に“typed”と言った場合は、プログラム中の変数や関数や式などに型がついている「**静的に型付けされた**」言語を表すことが多い。

JavaScript をはじめとする多くのスクリプト言語は、**変数**自体は型情報を持たず、**値**自身が型情報を保持している。これを「**動的に型付けされた**」言語ともいう。動的に型付けされた言語では、プログラムの実行過程に、ある変数の値の型が変わってゆくことがある。

## 2.4 簡単な関数型言語 FL の構文

基本型 (int, bool) とその演算 (四則演算や比較など) が与えられていると仮定する。すると

$$\begin{aligned}
 \text{(式)} \quad M ::= & c^\tau && (\text{con, 基本型の定数と演算}) \\
 & | x && (\text{var}) \\
 & | \text{nil} && (\text{nil}) \\
 & | M :: M && (\text{list}) \\
 & | \lambda x. M && (\text{abs, 関数抽象}) \\
 & | M M && (\text{app, 関数適用}) \\
 & | \text{if } M \text{ then } M \text{ else } M && (\text{if}) \\
 & | \text{let } x = M \text{ in } M && (\text{let}) \\
 & | \text{case } M \text{ of nil} \Rightarrow M | x :: x \Rightarrow M && (\text{case})
 \end{aligned} \tag{2.1}$$

ここで  $c^\tau$  というのは後述する型を表すが、自明ならば省略可能である。

FL のような (=基本型の追加などを施した) ラムダ計算の枠組みは、**応用ラムダ計算 (applied lambda calculus)** と呼ばれる。一方で、前半まで扱っていたラムダ計算は型という概念が無いので純粋ラムダ計算 (pure lambda calculus) や形なしラムダ計算 (untyped lambda calculus) とよばれる。これから扱うことの補足として、基本型に対する組込演算を  $Op$  とする。ここで  $Op$  は Curry 化された二項演算飲みからなるものとする。例えば加算というものは  $1 + 2$  と 2 変数を受け取るものであるが、 $f(x, y)$  と同じように考えれ

ばまず  $+$  を受け取ってから、変数の 1,2 を受け取るので

$$+ \ 5 \ 3 \rightarrow 8$$

となる。また同様にして、bool 型で返される、大小を判断する  $>$  という演算子も

$$> \ 4 \ 8 \rightarrow \text{false}$$

と表すことができる。

case について、簡単にいえば switch 文である。また nil は空のリストを表すので、 $M$  が空のリストの場合、 $N$  を選択し、var と var によるリストをみたら  $L$  を選択するというように考える。

## 2.5 FL の型

FL の型は前述した int 型と bool 型は基本型と呼ばれる。それ以外にも型はリスト型と関数型があり、以下のようにまとめて表現することができる。

$$\begin{array}{lcl} \tau ::= & \text{int} & \text{(基本型)} \\ & | \text{bool} & \text{(基本型)} \\ & | \text{list}(\tau) & \text{(リスト型)} \\ & | \tau \rightarrow \tau & \text{(関数型)} \end{array} \quad (2.2)$$

簡単な関数型言語 FL において、定数や組込演算の型はあらかじめ決まっているとする。たとえば、 $+$  演算子の型は 2 つの int 型を受け取って int 型を返す演算なので Curry 化すると  $\text{int} \rightarrow \text{int} \rightarrow \text{int}$  と表すことができる。また  $>$  演算子の型は 2 つの int 型を受け取って bool 型を返す演算なので、 $\text{int} \rightarrow \text{int} \rightarrow \text{bool}$  と表すことができる。

## 2.6 FL の small-step semantics

意味記述として 2 つの意味論があった。1 つめは**操作的**意味論 (operational semantics) である。これはプログラムの実行の形式化 (=“how” の形式化) に基づくプログラミング言語の意味記述方法である。具体的に抽象的な計算機<sup>\*9</sup>を定義し、プログラミング言語の意味を抽象的計算機の動作 (状態遷移) として記述する。しかし、この意味論はプログラミング言語の意味定義には不適當である。これに適しているのが**構造的**操作的意味論 (structural operational semantics) である。抽象構文にしたがった (=syntax-directed な) 帰納的定義をする方法である。

FL<sup>\*10</sup>で扱うのは small-step semantics である 1 ステップの簡約関係 “ $\rightarrow$ ” を推論規則の形で定義する。対比して big-step semantics<sup>\*11</sup>というものがある。部分の遷移から全体の遷移を定義する。ボトムアップにもトップダウンにも読むことができる。

まず app(関数適用) について以下の small-step semantics がある。

$$\begin{array}{ll} \text{(app1)} \quad \frac{M \rightarrow M'}{MN \rightarrow M'N} & \text{(app2)} \quad \frac{}{(\lambda x.M)N \rightarrow M[x := N]} \\ \text{(app3)} \quad \frac{M \rightarrow M'}{\text{op}M \rightarrow \text{op}M'} \quad (\text{if } \text{op} \in Op) & \text{(app4)} \quad \frac{N \rightarrow N'}{\text{opc } N \rightarrow \text{opc } N'} \quad (\text{if } \text{op} \in Op) \\ \text{(app5)} \quad \frac{}{\text{op } c_1 c_2 \rightarrow c} \quad (\text{if } \text{op}^{\tau_1 \rightarrow \tau_2 \rightarrow \tau} c_1^{\tau_1} c_2^{\tau_2} = c^\tau) \end{array}$$

$\rightarrow$  というのは簡約関係を表す。ここで、関数適用の引数や関数抽象の本体を簡約する規則がないことに注意する。

<sup>\*9</sup> 例としてオートマトンやチューニングマシン、ラムダ計算などがある。

<sup>\*10</sup> short for Function Level の略。

<sup>\*11</sup> どうして state がそんな変化をするかはあまり気にしない semantics. Natural semantics とする場合もある



(app1) というのはラムダ式の接続の左側は右側に影響することなく簡約することができる。

(app2) というのはラムダ抽象への右からの接続は関数適用であり、代入操作に簡約することができる。

(app3) というのは2項演算子がある時には、まずその後ろにある引数から簡約する。

(app4) というのは2項演算子の引数は第1引数が簡約されきって初めて第2引数を簡約する。

(app5) というのは定数同士の演算はその演算結果に簡約することができる。

この5つの規則から、FL では今まで扱っていた純粹ラムダ計算と異なり、必ず左から右に簡約が進んでいくということがわかる。

次に if, let を表す small-step semantics は以下のようなものである。

$$\begin{aligned}
 & \text{(if 1)} \quad \frac{L \rightarrow L'}{\text{if } L \text{ then } M \text{ else } N \rightarrow \text{if } L' \text{ then } M \text{ else } N} \\
 & \text{(if 2)} \quad \frac{}{\text{if true}^{\text{bool}} \text{ then } M \text{ else } N \rightarrow M} \\
 & \text{(if 3)} \quad \frac{}{\text{if false}^{\text{bool}} \text{ then } M \text{ else } N \rightarrow M} \\
 & \text{(let)} \quad \frac{}{\text{let } x = N \text{ in } M \rightarrow M[x := N]}
 \end{aligned}$$

(if 1) というのは if 文は条件式からまず簡約されることを表す。

(if 2) というのは if 文の条件式が true に簡約されたら、if 文全体は then 節に簡約されることを表す。

(if 3) というのは if 文の条件式が false に簡約されたら、if 文全体は else 節に簡約されることを表す。

(let) というのは let 文は  $L$  の中 (in) の  $x$  を  $M$  に置き換えるという意味である。

case については3つの small-step semantics は以下のようなものである。

$$\begin{aligned}
 & \text{(case 1)} \quad \frac{L \rightarrow L'}{\text{case } L \text{ of nil } \Rightarrow M \mid x :: y \Rightarrow N \rightarrow \text{case } L' \text{ of nil } \Rightarrow M \mid x :: y \Rightarrow N} \\
 & \text{(case 2)} \quad \frac{}{\text{case nil of nil } \Rightarrow M \mid x :: y \Rightarrow N \rightarrow M} \\
 & \text{(case 3)} \quad \frac{}{\text{case } L_1 :: L_2 \text{ of nil } \Rightarrow M \mid x :: y \Rightarrow N \rightarrow N[x \mapsto L_1][y \mapsto L_2]}
 \end{aligned}$$

(case 1) は switch 文の条件式 (マッチング対象) からまず簡約されることを表している。

(case 2) はマッチング対象が nil に簡約されたら、case 文全体は nil のときに置き換わるラムダ式に簡約されることを表している。

(case 3) はマッチング対象が  $H :: T$  に簡約されたら、case 文全体は  $h :: t$  のときに置き換わるラムダ式に簡約される。ただし、簡約時  $h$  は  $H$  に、 $t$  は  $T$  に置き換えられることを表している。

## 2.7 形式的体系

形式的体系とは今までラムダ計算には型がなかったが、この型をつけるための論理体系 (公理 + 推論規則) である。ここで型は安全性を保証する上で論ずる土台を与える。つまり型は安全性を完備する目的である。型体系では、以下の形の型判定式 (type judgment) を証明の対象とする。以下で表すことが出来る。

$$\Gamma \vdash M : \tau \quad (2.3)$$

- $\Gamma$  : 型環境 ( $M$  の中の自由変数の型を与える)
- $M$  : プログラム句 (phrase) (主部 (subject) ともいう。ラムダ計算ならラムダ式のこと)
- $\tau$  : 型

ここで、式 2.3 は  $\Gamma$  の下でプログラム  $M$  が型  $\tau$  をもつことを示している。

主に、 $\Gamma$  は型の事前条件が含まれる。この  $\Gamma$  が満たされているとき、 $M$  が型  $\tau$  をもつことを示すので、つまり型判定式は  $(x_1, x_2, x_3 \text{ は互いに異なる})$

$$x_1 : \tau_1, x_2 : \tau_2, x_3 : \tau_3 \vdash M : \tau$$



というのは「**論理式**  $x_1 : \tau_1 \wedge x_2 : \tau_2 \wedge x_3 : \tau_3 \Rightarrow M : \tau$  **が真である**」という意味である。

論理式や命題 (proposition) は真のことも偽のこともあるが判定式 (judgement) というのは、ある論理式や命題が真であることを説明するものである。

これから主に扱うのは型検査と型再構成 (type reconstruction) である。

**型検査**とは  $\Gamma, M, \tau$  が与えられたとき、型体系から  $\Gamma \vdash M : \tau$  を導く (証明する) ことが出来るかを調べることをいう。一方で**型再構成**は  $\Gamma, M$  が与えられたとき、型体系から  $\Gamma \vdash M : \tau$  を導けるような  $\tau$  を見つけることをいう。この型再構成は**型推論** (type inference) と呼ばれることもある。

ここから、FL の型体系についてまとめる。

$$\begin{array}{ll}
(\text{var}) \frac{}{\Gamma, x : \tau \vdash x : \tau} & (\text{abs}) \frac{\Gamma, x : \tau' \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : (\tau' \rightarrow \tau)} \\
(\text{con}) \frac{}{\Gamma \vdash c^\tau : \tau} & (\text{app}) \frac{\Gamma \vdash M : \tau' \rightarrow \tau \quad \Gamma \vdash N : \tau'}{\Gamma \vdash (MN) : \tau} \\
(\text{list}) \frac{\Gamma \vdash M_1 : \tau \quad \Gamma \vdash M_2 : \text{list}(\tau)}{\Gamma \vdash M_1 :: M_2 : \text{list}(\tau)} & (\text{let}) \frac{\Gamma \vdash N : \tau' \quad \Gamma, x : \tau' \vdash M : \tau}{\Gamma \vdash (\text{let } x = N \text{ in } M) : \tau} \\
(\text{nil}) \frac{}{\Gamma \vdash \text{nil} : \text{list}(\tau)} & \\
(\text{if}) \frac{\Gamma \vdash L : \text{bool} \quad \Gamma \vdash M : \tau \quad \Gamma \vdash N : \tau}{\Gamma \vdash (\text{if } L \text{ then } M \text{ else } N) : \tau} & \\
(\text{case}) \frac{\Gamma \vdash L : \text{list}(\tau_1) \quad \Gamma \vdash M : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \text{list}(\tau_1) \vdash N : \tau_2}{\Gamma \vdash (\text{case } L \text{ of nil} \Rightarrow M \mid x_1 :: x_2 \Rightarrow N) : \tau_2} &
\end{array}$$

(var) というのは、ある変数の型はその他の変数の型環境によらず独立に決定する。

(con) というのは、定数は他の型環境によらず暗黙的に型を持っている。

(list) というのは  $\tau$  型の要素を  $\tau$  のリストに接続したものもまた  $\tau$  型のリストである。

(nil) というのは、nil はリストであるが何型のリストであるということは規定されていない。

(if) というのは、条件式が bool 型で、then 節と else 節が同じ型なら if 文全体も then 節、else 節と同じ型を持つということを表している。

(abs) というのは、ラムダ抽象は、引数の方からラムダ式の型への関数型を与える。

(app) というのは、関数型に引数の型を食わせる関数適用は、返り値の型を与える。

(let) というのは、自由変数が  $\tau'$  型であるような型環境においては、 $\tau'$  のラムダ式をその自由変数に let 文で代入することができる。また let 文による代入は下のラムダ式の型を変えることはないことを表している。

(case) というのは、nil のときに置き換わるラムダ式が  $\tau'$  型で、 $h :: t$  のときに置き換わるラムダ式も  $\tau'$  型なら、case 文全体も  $\tau'$  型である。ただし、 $h :: t$  のときに置き換わるラムダ式は  $h, t$  をそれぞれ自由変数として持っている必要がある。さらに、マッチング対象が何らかリストで、 $h$  がそのリストの要素と同じ型で、 $t$  がそのリストと同じ型である必要がある。

## 索引

abstraction, 19  
Applicative order, 17  
  
bound variable, 5  
  
combinator, 6  
  
Diamond Property, 15  
  
equivalence relation, 8  
expansion, 8  
  
first-class citizen, 1  
free variable, 5  
  
lazy evaluation, 18  
leftmost, 17  
  
normal form, 7  
Normal order, 17  
  
outermost, 17  
  
scope, 3  
strong reduction, 6  
  
variable biding, 3  
variable capture, 10  
  
 $\eta$  簡約, 9  
 $\eta$  展開, 9  
 $\eta$  変換, 9  
  
応用ラムダ計算, 20  
  
Curry の不動点結合子, 13  
Curry-Howard 対応, 19  
  
結合子, 6  
原始再帰関数, 12  
  
構造的操作の意味論, 21  
後置演算子, 11  
合同関係, 9  
  
再帰関数, 12  
  
自由変数, 5  
純粋ラムダ計算, 20  
  
正規形, 7  
  
操作の意味論, 21  
束縛変数, 5  
  
遅延評価, 18  
チャーチ数, 12  
  
同値関係, 8  
閉じたラムダ式, 6  
  
標準化定理, 17  
  
 $\beta$  簡約, 6  
変数束縛, 3  
  
有効範囲, 3

## 参考文献

- [1] Computer Science Tripos(Part IB) Foundations of Functional Programming, <https://www.cl.cam.ac.uk/teaching/0607/FFuncProg/Founds-FP.pdf>
- [2] 第一級関数と第一級オブジェクト, <https://marycore.jp/coding/first-class-object-and-function/>
- [3] Ocamls, If 文, ループと再帰, [https://ocaml.org/learn/tutorials/if\\_statements\\_loops\\_and\\_recursion.ja.html](https://ocaml.org/learn/tutorials/if_statements_loops_and_recursion.ja.html)
- [4] プログラミング言語論 No. 4 : ラムダ計算, 高階の関数型言語, <http://www.cs.tsukuba.ac.jp/~kam/lecture/plm2018/4.pdf>
- [5] プログラミング言語論 No. 5 : , <http://www.cs.tsukuba.ac.jp/~kam/lecture/plm2018/5.pdf>
- [6] 『計算論理学』講義資料, <http://www.cs.tsukuba.ac.jp/~kam/complogic/main.pdf>
- [7] ラムダ計算入門, <http://www.kb.ecei.tohoku.ac.jp/~sumii/class/keisanki-software-kougaku-2005/lambda.pdf>
- [8] ラムダ計算 1, <http://abelard.flet.keio.ac.jp/person/takemura/class/old2/old/2008/print-lambda.pdf>
- [9] 言語モデル論, <https://www.ueda.info.waseda.ac.jp/oess/LM2011/>
- [10] 計算の理論, <http://www-kb.is.s.u-tokyo.ac.jp/~koba/class/ComputationTheory/ComputationTheory8.pdf>
- [11] 計算の理論「 $\lambda$  計算と型システム」, <http://www-kb.is.s.u-tokyo.ac.jp/~koba/class/ComputationTheory/ComputationTheory9.pdf>
- [12] ラムダ計算#3 マクロ, 不動点結合子と再帰関数, <http://kumachan-math.hatenablog.jp/entry/2019/01/30/004925>
- [13] ラムダ計算#4 FL と型体系, <http://kumachan-math.hatenablog.jp/entry/2019/01/31/140034>
- [14] ソフトウェア基礎論配布資料 (3)  $\lambda$  計算 (1), <https://www.fos.kuis.kyoto-u.ac.jp/~igarashi/class/sf04w/resume3.pdf>
- [15] Curry-Howard の対応, <http://www.cs.tsukuba.ac.jp/~kam/lecture/complogic2013/slide2013.pdf>
- [16] カリー・ハワード同型対応入門, <https://ocw.kyoto-u.ac.jp/ja/faculty-of-lettersja/002-006/pdf/curryhoward.pdf>
- [17] Lecture Notes on Types, <https://www.cl.cam.ac.uk/teaching/0910/Types/typ.pdf>
- [18] 型付き  $\lambda$  計算, [https://www.math.nagoya-u.ac.jp/~garrigue/lecture/2007\\_SS/typed.pdf](https://www.math.nagoya-u.ac.jp/~garrigue/lecture/2007_SS/typed.pdf)
- [19] 操作的意味論, <http://research.nii.ac.jp/~ichiro/lecture/model2003/operational%20semantics.pdf>
- [20] 型システム入門プログラミング言語と型の理論