

# 実践離散数学と計算の理論

## 5回目 $\lambda$ 計算

大阪大学大学院 工学研究科 電気電子情報通信工学専攻

王 イントウ

# 質問と回答

# 完全性に強弱はあるのでしょうか？

- ・ はい、あります。
- ・ 記号の定義とかある(下の参考資料をご参考)が、簡単に言えば、普通の(弱い)完全性の証明では空でない論理式集合(様々な公理や定理など)を用いて論理式を証明してるが、強い完全性の証明では空の論理式集合を用いても論理式を証明できる。
- ・ 強い完全性定理が成り立てば自動的に普通の完全性定理も成り立つ。
- ・ 参考資料：

<http://www.math.tsukuba.ac.jp/~tsuboi/gra/complete.pdf>

<https://scrapbox.io/mrsekut-p/%E5%AE%8C%E5%85%A8%E6%80%A7%E5%AE%9A%E7%90%86>

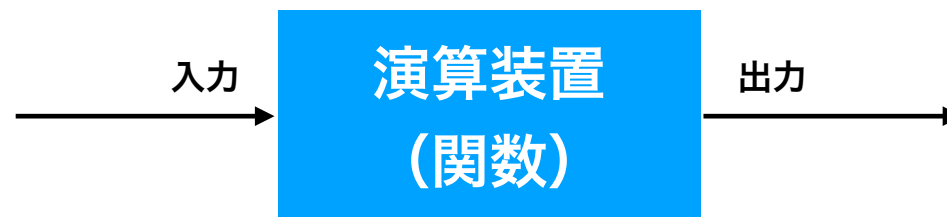
# $\lambda$ 計算 (Lambda Calculus)

# 初めに

- ・ チューリングマシンはチューリング完全と言って、どんなアルゴリズムでも実行できるような計算モデルである
- ・  $\lambda$  計算もチューリングマシンと同じように、どんなアルゴリズムでも実行できるような計算モデルとして知られている
- ・  $\lambda$  計算はプログラミング言語の基礎的な理論を構築するために使われることが多く、論理学から計算の方にやっていく、プログラミング言語に近いようになっている

# 関数 (Function)

- 関数とは、いくつか値を取り、なにか演算をして、演算した結果を出力する装置



$$x = 3, y = 2 \longrightarrow f(x, y) = x + y^2 + 4 \longrightarrow 11$$

# 一般的な関数の構成

- 関数は、名前、束縛変数、自由変数、定数、演算から成り立つ

fが関数の名前

$$f(\textcircled{x}, \textcircled{y}) = \textcircled{x} + \textcircled{y}^2 + \textcircled{C} + \textcircled{3}$$

束縛変数                      自由変数                      定数

# 関数適用 (Function Application)

- ・ 関数になにか値を入力することを、関数適用と呼ぶ

$$f(x, y) = x + y^2 + C + 3 \quad \text{関数定義 (function definition)}$$

$$f(3, 2) \quad \text{関数適用 (function application)}$$



# 束縛変数 (Bound Variable)

- ある関数があり、その関数に対して関数適用したときに、式中で置換される変数のことを束縛変数と呼ぶ

$$f(x, y) = x + y^2 + C + 3$$

という式があったときに  $f(3, 2)$  と関数適用すると、  
変数  $x, y$  が式中で置換される。

$$f(3, 2) \rightarrow 3 + 2^2 + C + 3$$

# 自由変数 (Free Variable)

- ・ 式の引数に束縛されずに、式の外側で自由に値を決めることの出来る変数を、自由変数と呼ぶ

$$f(x, y) = x + y^2 + \boxed{C} + 3$$

この変数Cは、関数fだけでは値が決められない

Eg. 重力定数(惑星より違う)、水の沸点(大気圧より違う)とか

# 簡約 (reduction)

- ・ 簡約とは、式に値を適用して、引数に対応する束縛変数を適用した値に置き換える操作のことである
- ・ すなわち、**計算とは簡約のことである**と捉えることが出来る

$$f(x, y) = x + y^2 + 3$$

$$f(3, 2) \rightarrow 3 + 2^2 + 3$$

# λ 計算 (Lambda Calculus)

- Alonzo Churchが1930年代に考案した計算モデル
- 計算能力は、チューリングマシンと等価であることが知られている
- Haskell、ML系言語、Lispなど多くの関数型言語の基礎となっている

# λ 計算の特徴

- 関数に名前がない（すべて無名関数）
- 関数の引数は1つだけ
- λ 計算のみで、条件分岐、再帰、四則演算が構成可能（チューリング完全）
- 原始帰納的関数は無限ループができないような関数の計算モデルであるが、λ 計算は無限ループができる

# λ 計算の構文 (Syntax)

$e$	$:=$	$v$	変数
		$\lambda v.e$	λ 抽象
		$(e\ e)$	関数適用

驚くほど単純だが、これだけで、アルゴリズムと呼べるものはすべて実行できる。

値：  $x := \lambda v.e$

# λ 抽象 (Lambda Abstraction)

- ・ 関数を λ という記号で表現したもので、つまるところ無名関数

$f(x, y) = x + y$  という数学の関数を λ 計算で書くと以下のようなになる。

$$\lambda x . \lambda y . (x + y)$$

(実際は λ 計算には加算は別の形で定義されているが、例えばとして)

# 関数適用 (Function Application)

- 関数適用は若干記述が違うが、数学の関数適用とほぼ同じ

例1  $f(x)=x$

$$(\lambda x . x \ 4) \rightarrow 4$$

例2 簡約

$$((\lambda x . \lambda y . (x \times y) \ 2) \ 3) \rightarrow (\lambda y . (2 \times y) \ 3) \rightarrow 2 \times 3$$



# カリー化 (Currying)

- ・ 複数の引数を取る関数を、1引数の関数で表現する方法
- ・ 引数がいくつあってもカリー化可能

$f(x, y) = x \times y$      2引数とする関数



カリー化

$\lambda x . \lambda y . (x \times y)$      1引数とする関数の組み合わせで表現

# 部分適用 (Partial Application)

- ・ カリー化することで、一部引数のみに関数適用して、新たな関数を得ることが可能

$\lambda x . \lambda y . (x \times y)$  2つの値を掛け算する関数



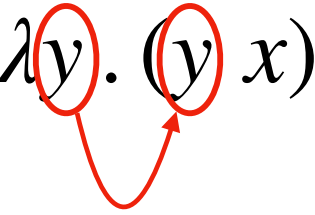
部分適用

$(\lambda x . \lambda y . (x \times y) \ 2) \rightarrow \lambda y . (2 \times y)$  ある値を2倍する関数

# 束縛変数 (Binding Variable)

- ・  $\lambda$  抽象の引数で束縛される変数
- ・ つまり、簡約すると置き換えられる変数

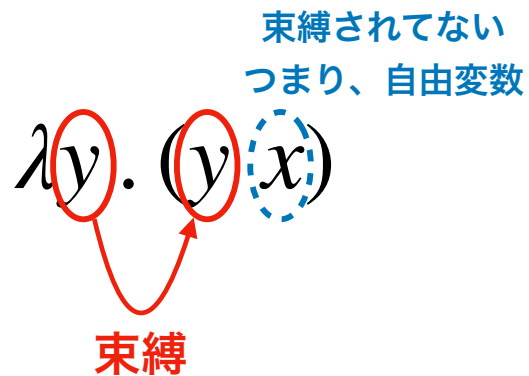
$\lambda y. (y x)$



束縛

# 自由変数 (Free Variable)

- ・  $\lambda$  抽象の引数に束縛されないような変数



# 変数束縛 (Variable Bindings)

- ・  $\lambda x . E$  という  $\lambda$  抽象があったとき、式  $E$  中の自由変数  $x$  が束縛される

$\lambda x . \lambda x . x$



この式中の自由変数  $x$  を束縛するが、  
自由変数がないため束縛される変数はない

# $\alpha$ 変換 (Alpha Conversion)

- ・ 束縛変数の変数名を変えても式の意味は変わらない
- ・ 束縛変数の変数名を置換することを  $\alpha$  変換と呼ぶ

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

束縛変数の名前をxからZに変えても意味は同じ

$$f(x) = x^2 + 5 \rightarrow f(Z) = Z^2 + 5$$

一般的な数学の関数でも同じ

- ・  $\lambda x . \lambda y . x$  の束縛変数  $x$  を  $y$  にして  $\lambda y . \lambda y . y$  とするのも  $\alpha$  変換ではない：  
内側のラムダ抽象  $\lambda y . x$  では  $x$  は自由変数だったのに  $\lambda y . y$  では自由変数ではなくなっている

# $\alpha$ 同値 (Alpha-equivalence)

- 2つの $\lambda$ 式 $E1$ と $E2$ があったとき、何度か $\alpha$ 変換を適用すると同じ式になるとき、 $E1$ と $E2$ は $\alpha$ 同値であると言う

$\lambda x . \lambda y . (x \ y)$  と  $\lambda a . \lambda b . (a \ b)$  は  $\alpha$  同値である

なぜなら

$$\lambda x . \lambda y . (x \ y) \rightarrow \lambda a . \lambda y . (a \ y) \rightarrow \lambda a . \lambda b . (a \ b)$$

# $\alpha$ 変換出来ない例

- 例えば、以下の式があったときに、束縛変数  $x$  を  $y$  に  $\alpha$  変換するとどうなるか？

$$\lambda x . (x + y) \rightarrow \lambda y . (y + y)?$$

元の式は、引数 $+y$ を行う式だったが、 $\alpha$ 変換後は引数を2倍する式になってしまっている

ここでは、 $+$ 演算子が仮に使えるとする



# $\alpha$ 変換の制約

- ・  $\lambda x . E$  という式があったときに、式  $E$  中に自由変数  $y$  が無いときにかぎり、束縛変数  $x$  を  $y$  に  $\alpha$  変換可能

$$\lambda x . (x + y)$$

この式中に自由変数  $y$  があるため、  
束縛変数  $x$  を  $y$  に置換できない

- ・ 述語論理の例：

# UI規則の失敗例

「すべての人間は母を持つ」

$x, y \in \text{人間}$        $y$  は  $x$  の母である  $:= P(x, y)$

$\forall x W(x) = \forall x \exists y P(x, y)$

$\exists y P(x, y)[x \mapsto y] = \exists y P(y, y)$

(間違ったUI規則の適用:  
 $y$ は $W(x)$ に束縛されるので)

「自分が自分の母であるような人間が存在する」

# 置換 (Substitution)

- ・ 式  $E$  中の自由変数  $x$  を  $y$  に置き換えるような操作を置換といい、 $[x \mapsto y]$  と表し、式  $E$  への適用を  $E[x \mapsto y]$  と表す

$$(\lambda x. (x \textcolor{red}{z})) [z \mapsto y] \rightarrow \lambda x. (x \ y)$$

$z$  は自由変数なので  $y$  に置換される

$$(\lambda \textcolor{red}{x}. (\textcolor{red}{x} \ z)) [x \mapsto y] \rightarrow \lambda x. (x \ z)$$

$x$  は束縛変数なので置換されない

# $\beta$ 簡約 (Beta Reduction)

- $(\lambda x.M N)$  という関数適用を行う式があったときに、 $\lambda x.M$  に値  $N$  を適用する操作を  $\beta$  簡約と呼ぶ
- $(\lambda x.M N)$  は、 $M[x \mapsto N]$  と考えられる

$$(\lambda x.(x + 3) 5) \rightarrow (x + 3)[x \mapsto 5] \rightarrow (5 + 3)$$

# 単純に $\beta$ 簡約できない例

- 変数名によっては、単純に  $\beta$  簡約出来ない場合がある

$$(\lambda y . \lambda x . (x + y) x) \rightarrow \lambda x . (x + x)?$$

これは式の意味が変わってしまってる

適用した式中の自由変数が新たに束縛されるので失敗

- $\beta$  簡約の前に、 $\alpha$  変換で束縛変数を置換

$$(\lambda y . \lambda x . (x + y) x) \rightarrow_{\alpha} (\lambda y . \lambda z . (z + y) x) \rightarrow_{\beta} \lambda z . (z + x) \quad (\text{ほしい結果})$$

# 簡約可能式 (Redex)

- ・ 簡約可能な式を簡約可能式 (redex) と呼ぶ
- ・ 特に、 $\beta$  簡約可能な式を、 $\beta$  基 ( $\beta$ -redex) と呼ぶこともある

$(\lambda x . x \ y)$

$\beta$  簡約でより簡単な式に簡約可能なので、これは簡約可能式 ( $\beta$  基) となる

# 評価戦略 (Evaluation Strategy)

- $(\lambda x.M N)$ という簡約可能式があったときに、次の2つの方法で評価する方法が考えられる
- 正規順序の評価 (normal-order evaluation)
  - 先に  $\beta$  簡約を行う
- 作用的順序の評価 (applicative-order evaluation)
  - 先に  $N$  の評価を行う

# 評価戦略の例

正規順序の評価

$$\begin{aligned} & (\lambda x.(x + x) \ (3 + 4)) \\ & \rightarrow (3 + 4) + (3 + 4) \\ & \rightarrow 7 + (3 + 4) \\ & \rightarrow 7 + 7 \\ & \rightarrow 14 \end{aligned}$$

- ・ 正規順序の場合、計算ステップが多くなる場合がある
- ・ Haskellはこれを効率よくした、必要呼出し (call-by-need) という方法をとっている
- ・ 無限のリストとかを簡単に扱える

作用的順序の評価

$$\begin{aligned} & (\lambda x.(x + x) \ (3 + 4)) \\ & \rightarrow (\lambda x.(x + x) \ 7) \\ & \rightarrow 7 + 7 \\ & \rightarrow 14 \end{aligned}$$

- ・ C、Python、Rubyなど、多くのプログラミング言語はこちら
- ・ 作用的順序の評価の方がコンピュータのhardwareの操作に近いので、わかりやすい



# 不動点 (Fixed Point)

- ・ 関数  $f :: a \rightarrow a$  に対して、 $f\ x == x$  となる  $x$  を「関数  $f$  の不動点」という。
- ・  $(\lambda x.(x\ x)\ \lambda x.(x\ x))$  という式は  $\beta$  簡約後も同じ式になり、このとき、 $\lambda x.(x\ x)$  は  $\lambda x.(x\ x)$  の不動点であるという

$$\begin{aligned} & (\lambda x.(x\ x)\ \lambda x.(x\ x)) \\ \rightarrow & (x\ x)[x \mapsto \lambda x.(x\ x)] \\ \rightarrow & (\lambda x.(x\ x)\ \lambda x.(x\ x)) \\ \rightarrow & \dots \end{aligned}$$

# 評価戦略の例その2

正規順序の評価

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

正規順序の場合、計算は停止する

作用的順序の評価

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

作用的順序の場合、必要のない式  
まで計算されてしまう（この例だ  
と計算が停まらなくなっていし  
まっている）

# 高階関数 (Higher-order Function)

- $\lambda$  計算は  $\lambda$  抽象 (関数) も第一級オブジェクトのため、引数に適用可能
- また、関数を返す関数も定義可能
- このような関数を引数にとったり、関数を返すような関数を高階関数と呼ぶ
- 例えば、不動点は関数を引数にとる関数であり、カリー化は関数を返す関数に変換する操作である

# 不動点コンビネータ (Fixed Point Combinator)

- $\lambda$  計算で再帰を行えるような関数として、不動点コンビネータと呼ばれる式が知られている
- 正規順序の評価の不動点コンビネータは、Yコンビネータ、作用的順序の評価の不動点コンビネータは、Zコンビネータと呼ばれる

$$Y \stackrel{def}{=} \lambda f. (\lambda x. (f (x x)) \quad \lambda x. (f (x x)))$$

$$Z \stackrel{def}{=} \lambda f. (\lambda x. (f \lambda y. ((x x) y) \quad \lambda x. (f \lambda y. ((x x) y)))$$

# Church数 (Church Encoding)

- 自然数を  $\lambda$  計算で表現する手法をChurch数と呼ぶ
- 条件分岐、数の表現、ループなど、アルゴリズムと呼ばれるものは、なんでも表すことができる  
(ノリ的には原始帰納的関数と同じ)

# 真偽値 (Boolean Value)

- ・ 真偽値は以下のように表現可能

$$\text{True} \stackrel{\text{def}}{=} \lambda x . \lambda y . x$$

$$\text{False} \stackrel{\text{def}}{=} \lambda x . \lambda y . y$$

# if式 (If Expression)

- ・ 条件分岐は以下のように定義可能

$$\text{if } c \text{ then } e_1 \text{ else } e_2 \stackrel{\text{def}}{=} (((\lambda x. \lambda y. \lambda z. ((x \ y) \ z) \ c) \ e_1 \ e_2)$$

# if式の例

$$\begin{aligned}\text{if True then } A \text{ else } B &= (((\lambda x.\lambda y.\lambda z.((x\ y)\ z)\ \text{True})\ A)\ B) \\ &\rightarrow ((\lambda y.\lambda z.((\text{True}\ y)\ z)\ A)\ B) \\ &\rightarrow (\lambda z.((\text{True}\ A)\ z)\ B) \\ &\rightarrow ((\text{True}\ A)\ B) = ((\lambda x.\lambda y.x\ A)\ B) \\ &\rightarrow (\lambda y.A\ B) \\ &\rightarrow A\end{aligned}$$

参考

$$\begin{aligned}\text{if } c \text{ then } e_1 \text{ else } e_2 &\stackrel{\text{def}}{=} (((\lambda x.\lambda y.\lambda z.((x\ y)\ z)\ c)\ e_1)\ e_2) \\ \text{True} &\stackrel{\text{def}}{=} \lambda x.\lambda y.x\end{aligned}$$



# let式 (Let Expression)

- 関数型言語で見られるlet式は以下のように定義可能

$$\text{let } v = e_1 \text{ in } e_2 \stackrel{\text{def}}{=} (\lambda v. e_2 \ e_1)$$

# let式の例

let  $x = A$  in

if  $x$  then  $e_1$  else  $e_2$

$= (\lambda x. (\text{if } x \text{ then } e_1 \text{ else } e_2) A)$

$\rightarrow \text{if } A \text{ then } e_1 \text{ else } e_2$

参考

let  $v = e_1$  in  $e_2 \stackrel{\text{def}}{=} (\lambda v. e_2 e_1)$

# 整数値 (Integer Value)

- 0と後継者関数 (successor) は以下のように定義可能

$$0 \stackrel{def}{=} \lambda f. \lambda x. x$$

$$\text{succ}(n) \stackrel{def}{=} \lambda n. \lambda f. \lambda x. (f ((n f) x))$$

$f$  は原始帰納的関数のsucc()からカリー化したもの

# 整数値の例

$$\begin{aligned} 1 = \text{succ}(0) &= (\lambda n. \lambda f. \lambda x. (f ((n f) x))) 0 \\ &\rightarrow \lambda f. \lambda x. (f ((0 f) x)) = \lambda f. \lambda x. (f ((\lambda f. \lambda x. x f) x)) \\ &\rightarrow \lambda f. \lambda x. (f (\lambda x. x x)) && (\text{ここでは、関数内を内部評価している}) \\ &\rightarrow \lambda f. \lambda x. (f x) \end{aligned}$$

$$\begin{aligned} 2 = \text{succ}(1) &= (\lambda n. \lambda f. \lambda x. (f ((n f) x))) 1 \\ &\rightarrow \lambda f. \lambda x. (f ((1 f) x)) = \lambda f. \lambda x. (f ((\lambda f. \lambda x. (f x) f) x)) \\ &\rightarrow \lambda f. \lambda x. (f ((\lambda x. (f x) x))) \\ &\rightarrow \lambda f. \lambda x. (f (f x)) \end{aligned}$$

$$\begin{aligned} \text{succ}(n) &\stackrel{\text{def}}{=} \lambda n. \lambda f. \lambda x. (f ((n f) x)) \\ 0 &\stackrel{\text{def}}{=} \lambda f. \lambda x. x \end{aligned}$$

# 整数値まとめ

整数値は、関数 $f$ を $x$ に $n$ 回適用するような式となる

$$1 = \lambda f. \lambda x. (f\ x)$$

$$2 = \lambda f. \lambda x. (f\ (f\ x))$$

$$3 = \lambda f. \lambda x. (f\ (f\ (f\ x)))$$

$$4 = \lambda f. \lambda x. (f\ (f\ (f\ (f\ x))))$$

●  
●  
●

# 加算 (Addition)

- ・ 加算は以下のように定義可能

$$v_1 + v_2 \stackrel{def}{=} ((\lambda m . \lambda n . ((m \textit{ succ}) n) v_1) v_2)$$

レポート

# 問題1

Church数でエンコードされた以下の $\lambda$ 式が、  
最終的にChurch数の3に簡約されることを1ステップずつ簡約して示せ  
(必要な場合は、関数内を部分評価せよ)

$\text{let } x = 2 \text{ in } 1 + x$



# 問題2

- ・  $\lambda$  計算をベースとして、簡易なプログラミング言語を自由に設計せよ(定義、値を代入で検証)
- ・ 構文を明確にすること
  - ・ 条件分岐、関数定義、再帰関数定義、引数が複数ある関数の定義、変数定義、 $+$ 、 $-$ 、 $*$ 、 $/$ 、ループなど
- ・ 上記構文と、 $\lambda$  式への対応を明確にすること

# レポート課題

- ・ 本スライド中にある問題を解いてレポートとして提出せよ
- ・ 締め切り：2023年7月11日 23:50 (JST)