

# 工学部専門科目「計算と論理」配布資料

## 直観主義論理とカリー・ハワード同型対応

五十嵐 淳

京都大学 大学院情報学研究科 通信情報システム専攻

cal15@fos.kuis.kyoto-u.ac.jp

<http://www.fos.kuis.kyoto-u.ac.jp/~igarashi/class/cal/>

January 12, 2016

### 1 直観主義論理

既に簡単にふれたように、Coq の論理体系および講義でふれた自然演繹で与えた論理体系はいずれも直観主義論理 (*intuitionistic logic*) と呼ばれるもので、命題の意味を (真/偽を表す)1/0 で与えるふつうの論理 (これを古典論理 (*classical logic*) と呼ぶ) とは異なるものである。

直観主義論理の特徴的な原理は

- 「 $A$  または  $B$  である」という命題が正しいと主張するためには、 $A$  と  $B$  のどちらが正しいのかを述べないといけない
- 「性質  $P$  を満たす  $x$  が存在する」という命題が正しいと主張するためには、実際に  $P$  を満たす具体的な  $x$  を示す必要がある

ということにある。これらのことは当たり前に見えるかもしれないが、これらの原理の裏側に隠れているのは、

- 「 $A$  でもなく、 $B$  でもないようなことはない ( $\neg(\neg A \wedge \neg B)$ )」ことが示せたからといって、それだけでは「 $A$  または  $B$  である」といってはいけない。(「一体どっちが成立しているんだよ!」というツッコミが入る。)
- 「どの  $x$  に対しても  $P$  ではない、というわけではない ( $\neg(\forall x, \neg P[x])$ )」ことが示せたからといって、(それだけで)「性質  $P$  を満たす  $x$  が存在する」といってはいけない。(「じゃあその  $x$  とやらを教えてくれ!」というツッコミが入る。)

ということである。例えば「円周率には 7 が 7 連続で出現する場所があるかないかのいずれかである」と直観主義者に主張するためには、まず「ある」か「ない」かをはっきりさせ、「ある」のであれば何桁目からそれが出現するかも含めて示さないと<sup>1</sup>、直観主義者を納得させることはできない。上の円周率の例からもわかるように、これらの原理は、排中律

---

<sup>1</sup> 「ない」のであれば、7 連続で出現したら矛盾することを示さないと

どんな命題  $P$  でも、 $P$  または  $\neg P$  いずれかが成立する

は認めない、という立場にもつながる。

また、直観主義の考え方を表す、もうひとつの有名な例は

「無理数  $x, y$  で  $x^y$  が有理数であるようなものがある」

という主張に対する証明である。この主張に対する簡単な証明は以下のようなものである。

$\sqrt{2}^{\sqrt{2}}$  が有理数ならば、 $x = y = \sqrt{2}$  とすればよい。 $\sqrt{2}^{\sqrt{2}}$  が無理数ならば、 $x = \sqrt{2}^{\sqrt{2}}, y = \sqrt{2}$  とすればよい。

このような「証明」を直観主義者は認めない。なぜだろうか。<sup>2</sup>

その他、古典論理では認められるが、直観主義の立場では認められない命題には以下のようなものがある。

- 二重否定除去:  $(\neg\neg P) \rightarrow P$
- パース則<sup>3</sup>:  $((P \rightarrow Q) \rightarrow P) \rightarrow P$
- 排中律:  $P \vee \neg P$ 
  - ただし、 $\neg\neg(P \vee \neg P)$  は成立
- ド・モルガン則<sup>4</sup>(の一部):  $\neg(\neg P \wedge \neg Q) \rightarrow P \vee Q$
- $(P \rightarrow Q) \rightarrow (\neg P \vee Q)$

しかし、このような弱い(「正しい」と認められることが少ない)論理を考える意義は何だろうか。それは、まさに上で述べた特徴にある。つまり直観主義論理で行った「 $P$ を満たす  $x$ が存在する」という存在証明からは、単に「何かが存在する」という情報だけでなく、 $P$ を満たす具体的な  $x$  が得られるという利点がある。(逆に、上でみた「無理数  $x, y$  で  $x^y$  が有理数であるようなものがある」の簡単な証明を見ても  $x, y$  が具体的に何かわからない。)また、後で述べるように、直観主義の立場では、「任意の  $x$  に対して  $P(x)$ 」という主張は、与えられた  $x$  から  $P(x)$  の証明が計算できることによって認められる。よって、「任意の  $x$  に対して、ある  $y$  が存在して～」という形の主張の証明は、 $x$  から「ある  $y$  が存在して～」の証明を計算する方法があることになる。「ある  $y$  が存在して～」の証明には、「～」を見たす  $y$  が含まれているのであるから、 $x$  から  $y$  を計算する方法が証明中に埋め込まれているともいえる。例えば素数が無限にある、という主張

任意の素数  $x$  に対し、 $y > x$  なる素数  $y$  がある

という主張に対する(直観主義の立場で認められる)証明には、与えられた素数からそれより大きい素数の計算方法が書いてある、ということになる。

<sup>2</sup>ちなみに、この主張に対する直観主義者でも認められる証明もある。わかりますか？

<sup>3</sup>パース (Charles Sanders Peirce, 1839–1914) はアメリカの論理学者・数学者・哲学者。

<sup>4</sup>ド・モルガン (Augustus de Morgan, 1806–1871) は(インド生まれの)イギリスの数学者。

## 2 カリー・ハワード同型対応

カリー・ハワード同型対応 (*Curry-Howard isomorphism*)<sup>5</sup> は、論理の証明体系と型付の計算体系における諸概念が対応している、という考え方である。この際、対応の鍵となるのが「命題を型として捉える」もしくは「導出をプログラムとして捉える」という観点である。この考えは、Coq などの多くの証明支援系の設計の原理となっている。

### 2.1 「ならば」の論理 vs. 単純型付ラムダ計算

まずは、左の「ならば」の論理の導出規則と右の単純型付ラムダ計算の型付け規則を比べてもらいたい。

$$\begin{array}{c}
 \frac{(H : P \in \Gamma)}{\Gamma \vdash P} \quad (\text{ASSUMPTION}) \\
 \\
 \frac{\Gamma, H : P \vdash Q \quad (H \notin \text{dom}(\Gamma))}{\Gamma \vdash P \rightarrow Q} \quad (-\rightarrow\text{I}) \\
 \\
 \frac{\Gamma \vdash P \rightarrow Q \quad \Gamma \vdash P}{\Gamma \vdash Q} \quad (-\rightarrow\text{E})
 \end{array}
 \qquad
 \begin{array}{c}
 \frac{(x : T \in \Gamma)}{\Gamma \vdash x : T} \quad (\text{T-VAR}) \\
 \\
 \frac{\Gamma, x : S \vdash M : T \quad (x \notin \text{dom}(\Gamma))}{\Gamma \vdash \text{fun } x : S \Rightarrow M : S \rightarrow T} \quad (\text{T-FUN}) \\
 \\
 \frac{\Gamma \vdash M : S \rightarrow T \quad \Gamma \vdash N : S}{\Gamma \vdash M N : T} \quad (\text{T-APP})
 \end{array}$$

型付け規則の判断から「 $M :$ 」部分を除けば、実質的に「ならば」の論理の導出規則になることがわかるだろう。原子命題を `nat` や `bool` の代わりに基本型だとし、項の種類を変数、`fun`、関数適用に限った単純型付ラムダ計算を考えると、ある命題が証明できることと、(命題を型として読みかえた時) その型を持つ項が存在することが必要十分になっていることがわかる。

これがカリー・ハワード同型対応の最も基本的な部分である。

論理	計算
命題	型
「ならば」	関数型 ( $\rightarrow$ )
文脈	型環境
仮定の名前	変数
(証明の) 導出規則	型付け規則
証明可能	その型を持つ項が存在する

### 2.2 プログラムは証明である

上の対応は単純型付ラムダ計算から項(つまりプログラム)を「無視する」ことで得られたが、論理体系において項に相当するものはあるのだろうか？実は、それは証明(つまり導出)である。

その感覚をつかむには以下の事実がヒントとなる。

<sup>5</sup>この名前は対応を発見・展開した(主な)人物である Haskell B. Curry (1900–1982) と William A. Howard (b.1926) に由来している。

**事実 1** 単純型付ラムダ計算において型付け関係  $\Gamma \vdash M : T$  が導出できるならば、その導出は  $\Gamma$ ,  $M$  から一意に定まる。

これは、項の形それぞれに、型付け規則がひとつ用意されていることから明らかである。特に、 $M$  を見れば、規則がどういう順番で使われるか決定される、ということである。つまり、単純型付ラムダ計算の項は、型付け関係の導出をコード化したものであり、導出を生成するいわばDNAの役割ができるのである。型付け関係の導出は(項を無視することで)命題の証明になるのであるから、項は命題の証明(導出)をプログラムの形で書いたもの、とすることができる。逆に、命題の証明(導出)から、項を得ることもできる。

すなわち

論理	計算
⋮	⋮
証明(導出)	項

ということである。例えば

「 $A$  ならば  $A$ 」の証明は `fun x:A => x` というプログラムである

ことになる。しかし、これは単なる記号法の類似なのだろうか。プログラムにおいて本質的な「計算」(簡約)は論理においてどんな役割があるというのだろうか。

この疑問はひとまず脇に置いておいて、対応関係が極小論理だけでなく、適当な型をラムダ計算に導入することで命題論理へと拡張できることを見る。

### 2.3 対応の命題論理への拡張

論理結合子「かつ」に対応する型は「ペア」(直積型)である。

(types)  $S, T ::= \dots \mid S * T$

(terms)  $M, N ::= \dots \mid (M_1, M_2) \mid \text{fst } M \mid \text{snd } M$   
 $\mid \text{match } M \text{ with } (x, y) \Rightarrow N \text{ end}$

$\text{fst } (M_1, M_2) \longrightarrow M_1$  (R-FST)

$\text{snd } (M_1, M_2) \longrightarrow M_2$  (R-SND)

$\text{match } (M_1, M_2) \text{ with } (x, y) \Rightarrow N[x, y] \text{ end} \longrightarrow N[M_1, M_2]$  (R-PMATCH)

$$\frac{\Gamma \vdash M : S \quad \Gamma \vdash N : T}{\Gamma \vdash (M, N) : S * T}$$
 (T-PAIR)

$$\frac{\Gamma \vdash M : S * T}{\Gamma \vdash \text{fst } M : S}$$
 (T-FST)

$$\frac{\Gamma \vdash M : S * T}{\Gamma \vdash \text{snd } M : T} \quad (\text{T-SND})$$

$$\frac{\Gamma \vdash M : T_1 * T_2 \quad \Gamma, x : T_1, y : T_2 \vdash N : S}{\Gamma \vdash \text{match } M \text{ with } (x, y) \Rightarrow N \text{ end} : S} \quad (\text{T-PMATCH})$$

論理結合子「または」に対応する型は直和型と呼ばれる型であり、 $S + T$  型の値は  $S$  の値か  $T$  の値のいずれか (に、どちら由来の値かを示すタグをつけたもの) となる。Coq で書くと以下のような型定義になり

```
Inductive sum (X Y : Type) : Type :=
| left : X -> sum X Y
| right : Y -> sum X Y.
```

例えば、`left nat bool 2` や `right nat bool true` はいずれも `sum nat bool` 型を持つ。sum 型の値を使う時には、真偽値やリストなどと同様に、`match` でタグの場合分けをすることになる。

```
(types) S, T ::= ...
          | S + T
```

```
(terms) M, N ::= ...
           | leftS,T M (型引数は重要な場合に添字として表記する)
           | rightS,T M
           | match M with left x => N1 | right y => N2 end
```

計算規則:

```
match left M1 with left x => N1[x] | right y => N2 end → N1[M1]
match right M2 with left x => N1 | right y => N2[y] end → N2[M2]
```

型付け規則:

$$\frac{\Gamma \vdash M : S}{\Gamma \vdash \text{left}_{S,T} M : S + T} \quad (\text{T-LEFT})$$

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash \text{right}_{S,T} M : S + T} \quad (\text{T-RIGHT})$$

$$\frac{\Gamma \vdash M : S_1 + S_2 \quad \Gamma, x : S_1 \vdash N_1 : T \quad \Gamma, y : S_2 \vdash N_2 : T \quad (x, y \notin \text{dom}(\Gamma))}{\Gamma \vdash \text{match } M \text{ with left } x \Rightarrow N_1 \mid \text{right } y \Rightarrow N_2 \text{ end} : T} \quad (\text{T-SMATCH})$$

まとめ:

論理	計算
⋮	⋮
「かつ」	直積型
「または」	直和型
導入規則	値の構築 (コンストラクタ, または fun)
除去規則	値の使用 (match または関数適用)

## 2.4 証明の回り道と証明の正規化

一般にひとつの命題の証明には何通りもの方法があるが、証明の中には無駄な推論ステップを含んでいるものがある。例えば、

$$\frac{\frac{\frac{\vdots}{\Gamma \vdash A} \quad \frac{\vdots}{\Gamma \vdash B}}{\Gamma \vdash A * B} \wedge\text{-I}}{\Gamma \vdash B} \wedge\text{-E2}$$

という導出を考える。この最終的な結論は  $\Gamma \vdash B$  だが、それは既に上で得られているものであり、最後の  $\wedge\text{-I}$ ,  $\wedge\text{-E2}$  の2ステップは無駄である。これが証明の回り道 (*detour*) である。そして、この回り道を除去し

$$\frac{\vdots}{\Gamma \vdash B}$$

という導出を得ることを、証明の正規化 (*normalization*) と呼ぶ。

この過程を、単純型付ラムダ計算の型付け関係の導出に移してみると

$$\frac{\frac{\frac{\vdots}{\Gamma \vdash M_1 : A} \quad \frac{\vdots}{\Gamma \vdash M_2 : B}}{\Gamma \vdash (M_1, M_2) : A * B} \text{T-PAIR}}{\Gamma \vdash \text{snd} (M_1, M_2) : B} \text{T-SND} \longrightarrow \frac{\vdots}{\Gamma \vdash M_2 : B}$$

と捉えられる。導出は、結論に現れる項と実質同じなので、この図から項を取り出してみると、

$$\text{snd} (M_1, M_2) \longrightarrow M_2$$

となり、これは簡約規則である!

つまり、

論理	計算
⋮	⋮
証明の回り道	簡約基 (簡約規則の左辺のこと)
証明の正規化	簡約 (プログラムの実行)

という対応が得られる。

この対応は「かつ」と直積型だけに留まらない。回り道はいずれも、ある論理結合子を導入した直後に除去している場合に対応する。例えば「ならば」に関する回り道は

$$\frac{\frac{\overline{\Gamma, x : S, \dots \vdash x : S} \quad \dots \quad \overline{\Gamma, x : S, \dots \vdash x : S} \quad \text{T-VAR}}{\Gamma, x : S \vdash M[x] : T} \quad \frac{\Gamma \vdash \text{fun } x : S \Rightarrow M[x] : S \rightarrow T \quad \text{T-ABS}}{\Gamma \vdash (\text{fun } x : S \Rightarrow M[x]) N : T} \quad \frac{\Gamma \vdash N : S \quad \text{T-APP}}{\Gamma \vdash (\text{fun } x : S \Rightarrow M[x]) N : T}}$$

という形である。残念ながら二段上の証明は文脈が  $x : S$  の分だけ違うので、「かつ」の場合と違い、この部分だけを切り出してきてもだめである。 $S$  の証明を仮定の使用 (これは変数参照である!) に「接木」したような導出にすれば  $x : S$  という仮定なしの証明が作れる。

$$\frac{\Gamma, \dots \vdash N : S \quad \dots \quad \Gamma, \dots \vdash N : S}{\Gamma \vdash M[N] : T}$$

という形の正規化が可能である。

また、「または」についての回り道 (のひとつ) は、

$$\frac{\frac{\Gamma \vdash M : S_1 \quad \text{T-LEFT}}{\Gamma \vdash \text{left } M : S_1 + S_2} \quad \frac{\overline{\Gamma, x : S, \dots \vdash x : S} \quad \dots \quad \overline{\Gamma, x : S, \dots \vdash x : S} \quad \text{T-VAR}}{\Gamma, x : S_1 \vdash N_1[x] : T} \quad \frac{\Gamma, y : S_2 \vdash N_2[y] : T \quad \text{T-SMATCH}}{\Gamma \vdash \text{match left } M \text{ with left } x \Rightarrow N_1[x] \mid \text{right } y \Rightarrow N_2 \text{ end} : T}}$$

であり、回り道の除去により、T-VAR を使った葉の部分に継木をした

$$\frac{\Gamma, \dots \vdash M : S_1 \quad \dots \quad \Gamma, \dots \vdash M : S_1}{\Gamma \vdash N_1[M] : T}$$

これらのことから、「 $A$  ならば  $B$ 」の証明は関数である、というのは「 $A$  の証明を受け取ると (回り道を取り除いた)  $B$  の証明を返す関数である」という意味であることがわかる。そして、「 $A$  かつ  $B$ 」の証明は「 $A$  の証明と  $B$  の証明の組」であり、「 $A$  または  $B$ 」の証明は「left というタグのついた  $A$  の証明」もしくは「right というタグのついた  $B$  の証明」なのである。

## 2.5 全称量子化と依存関数型

最後に全称量子化に対応する計算体系の側の概念である依存関数型 (*dependent function type*) について、大雑把にふれておく。

全称量子化  $\forall x : T, P$  に関する規則をよく見ると、「ならば」の規則とよく似ていることがわかる。実は、全称量子化  $\forall x : T, P[x]$  の証明も「ならば」の証明が関数であるのと同様に関数であると考えることができる。ただし、「ならば」の証明が証明を受け取る関数であるのとは違って「 $T$  型の値

$M$  を受け取って  $P[M]$  の証明を返す関数」となる。(MoreLogic.v で証明した `eq_nat_dec` は依存関数型の例となる。)

依存関数型は、まず型情報を詳細化することから現れてくる。例えば、自然数リストの型情報にその長さ(要素の数)を含めることを考えてみる。つまり、

- `nil` の型は `natlist(0)`
- `cons 3 nil` の型は `natlist(1)`
- `cons 1 (cons 2 (cons 3 nil))` の型は `natlist(3)`

のような感じである。すると、0 を  $n$  個並べたリストを返す関数 `makezeros` の型はどうなるだろうか。まず、引数の型は `nat` である。戻り値の型はどうなるだろうか。 $n$  を受け取った時の戻り値は

$$\underbrace{\text{cons } 0 \text{ (cons } 0 \text{ ... (cons } 0 \text{ nil)...)}}_n$$

なのだから、その型は引数の値によって異なる `natlist( $n$ )` になる。このような、引数の値に依存して戻り値の型が変わる関数の型を表すのが依存関数型である。依存関数型は一般に

$$\Pi x : S. T[x]$$

という形をしており、引数  $x$  の型が  $S$  で戻り値の型が  $T$  である(ただし、 $T$  には引数  $x$  が現れる)ことを表している。例えば、`makezero` の型は

$$\Pi n : \text{nat}. \text{natlist}(n)$$

となる。

依存関数型  $\Pi x : S. T[x]$  を持つ関数を引数  $M : S$  に適用すると、その結果の型は  $T[M]$  となる。例えば、

$$\begin{aligned} &\text{makezero } 3 : \text{natlist } 3 \\ &\text{makezero } (n + m) : \text{natlist}(n + m) \end{aligned}$$

などとなる。

単純な関数型  $S \rightarrow T$  は、実は依存関数型のうち、引数が戻り値の型に現れない(戻り値の型が引数に依存しない)特殊ケースの略記である。例えば、自然数を二倍する関数 `double` の型は、 $\Pi n : \text{nat}. \text{nat}$  (つまり、自然数  $n$  を受け取り、その値に関わらず自然数を返す、という意味である)とも考えられる。

### 3 Coq にみるカーリー・ハワード同型対応

- 定理の名前を `Check` コマンドにかけてみよう!
- 定義された名前を引数にとり定義内容を表示する `Print` コマンドに定理の名前を与えてみよう!
  - 「ならば」や「任意の～」の証明が関数になっていることを確認しよう
  - 等号についての証明の詳細はあまり気にしないこと!
  - 証明が再帰関数になっている (`fix` を使っている) のはどのような時だろうか。