工学部専門科目「計算と論理」配布資料
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
cal16@fos.kuis.kyoto-u.ac.jp
http://www.fos.kuis.kyoto-u.ac.jp/~igarashi/class/cal/

January 24, 2017

カリー・ハワード同型対応 (*Curry–Howard isomorphism*)[1] 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．

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．

$$\frac{(\boldsymbol{H} : \boldsymbol{P} \in \boldsymbol{\Gamma})}{\boldsymbol{\Gamma} \vdash \boldsymbol{P}} \quad (\textsc{Assumption}) \qquad \frac{(\boldsymbol{x} : \boldsymbol{T} \in \boldsymbol{\Gamma})}{\boldsymbol{\Gamma} \vdash \boldsymbol{x} : \boldsymbol{T}} \quad (\text{T-}\textsc{Var})$$

$$\frac{\boldsymbol{\Gamma},\ \boldsymbol{H} : \boldsymbol{P} \vdash \boldsymbol{Q} \qquad (\boldsymbol{H} \notin \mathit{dom}(\boldsymbol{\Gamma}))}{\boldsymbol{\Gamma} \vdash \boldsymbol{P}\ \texttt{->}\ \boldsymbol{Q}} \quad (\texttt{->}\text{I}) \qquad \frac{\boldsymbol{\Gamma},\ \boldsymbol{x} : \boldsymbol{S} \vdash \boldsymbol{M} : \boldsymbol{T} \qquad (\boldsymbol{x} \notin \mathit{dom}(\Gamma))}{\boldsymbol{\Gamma} \vdash \texttt{fun}\ \boldsymbol{x} : \boldsymbol{S}\ \texttt{=>}\ \boldsymbol{M} : \boldsymbol{S}\ \texttt{->}\ \boldsymbol{T}} \quad (\text{T-}\textsc{Fun})$$

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{P}\ \texttt{->}\ \boldsymbol{Q} \qquad \boldsymbol{\Gamma} \vdash \boldsymbol{P}}{\boldsymbol{\Gamma} \vdash \boldsymbol{Q}} \quad (\texttt{->}\text{E}) \qquad \frac{\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{S}\ \texttt{->}\ \boldsymbol{T} \qquad \boldsymbol{\Gamma} \vdash \boldsymbol{N} : \boldsymbol{S}}{\boldsymbol{\Gamma} \vdash \boldsymbol{M}\ \boldsymbol{N} : \boldsymbol{T}} \quad (\text{T-}\textsc{App})$$

型付け規則の判断から「$\boldsymbol{M}$:」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` の代わりの基本型だとし，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な部分である．

[1] 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．

論理	計算
命題	型
「ならば」	関数型 (->)
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 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．

事実 1 単純型付ラムダ計算において型付け関係 $\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{T}$ が導出できるならば，その導出は $\boldsymbol{\Gamma}$, $\boldsymbol{M}$ 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，$\boldsymbol{M}$ 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，と言うことができる．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．

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．

3 対応の命題論理への拡張

連言と直積型

論理結合子「かつ」に対応する型は「ペア」(直積型) である．

$$\text{(types)}\quad \boldsymbol{S}, \boldsymbol{T} \quad ::= \quad \cdots \mid \boldsymbol{S} * \boldsymbol{T}$$

$$\begin{array}{rrll} \text{(terms)} & \boldsymbol{M}, \boldsymbol{N} & ::= & \cdots \mid (\boldsymbol{M_1}, \boldsymbol{M_2}) \mid \texttt{fst}\ \boldsymbol{M} \mid \texttt{snd}\ \boldsymbol{M} \\ & & \mid & \texttt{match}\ \boldsymbol{M}\ \texttt{with}\ (\boldsymbol{x}, \boldsymbol{y})\ \texttt{=>}\ \boldsymbol{N}\ \texttt{end} \end{array}$$

$$\texttt{fst}\ (\boldsymbol{M_1}, \boldsymbol{M_2}) \longrightarrow \boldsymbol{M_1} \qquad \text{(R-Fst)}$$

$$\texttt{snd}\ (\boldsymbol{M_1}, \boldsymbol{M_2}) \longrightarrow \boldsymbol{M_2} \qquad \text{(R-Snd)}$$

$$\texttt{match}\ (\boldsymbol{M_1}, \boldsymbol{M_2})\ \texttt{with}\ (\boldsymbol{x}, \boldsymbol{y})\ \texttt{=>}\ \boldsymbol{N}[\boldsymbol{x}, \boldsymbol{y}]\ \texttt{end} \longrightarrow \boldsymbol{N}[\boldsymbol{M_1}, \boldsymbol{M_2}] \qquad \text{(R-PMatch)}$$

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{S} \qquad \boldsymbol{\Gamma} \vdash \boldsymbol{N} : \boldsymbol{T}}{\boldsymbol{\Gamma} \vdash (\boldsymbol{M}, \boldsymbol{N}) : \boldsymbol{S} * \boldsymbol{T}} \qquad \text{(T-Pair)}$$

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{S} * \boldsymbol{T}}{\boldsymbol{\Gamma} \vdash \texttt{fst}\ \boldsymbol{M} : \boldsymbol{S}} \qquad \text{(T-Fst)}$$

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{S} * \boldsymbol{T}}{\boldsymbol{\Gamma} \vdash \texttt{snd}\ \boldsymbol{M} : \boldsymbol{T}} \qquad \text{(T-Snd)}$$

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{T_1} * \boldsymbol{T_2} \qquad \boldsymbol{\Gamma}, \boldsymbol{x} : \boldsymbol{T_1}, \boldsymbol{y} : \boldsymbol{T_2} \vdash \boldsymbol{N} : \boldsymbol{S}}{\boldsymbol{\Gamma} \vdash \texttt{match}\ \boldsymbol{M}\ \texttt{with}\ (\boldsymbol{x}, \boldsymbol{y})\ \texttt{=>}\ \boldsymbol{N}\ \texttt{end} : \boldsymbol{S}} \qquad \text{(T-PMatch)}$$

選言と直和型

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，$\boldsymbol{S} + \boldsymbol{T}$ 型の値は $\boldsymbol{S}$ の値か $\boldsymbol{T}$ 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．

$$\begin{array}{rrll} \text{(types)} & \boldsymbol{S}, \boldsymbol{T} & ::= & \cdots \\ & & \mid & \boldsymbol{S} + \boldsymbol{T} \end{array}$$

$$\begin{array}{rrll} \text{(terms)} & \boldsymbol{M}, \boldsymbol{N} & ::= & \cdots \\ & & \mid & \texttt{left}_{\boldsymbol{S},\boldsymbol{T}}\ \boldsymbol{M} \quad \text{(型引数は重要な場合に添字として表記する)} \\ & & \mid & \texttt{right}_{\boldsymbol{S},\boldsymbol{T}}\ \boldsymbol{M} \\ & & \mid & \texttt{match}\ \boldsymbol{M}\ \texttt{with left}\ \boldsymbol{x}\ \texttt{=>}\ \boldsymbol{N_1}\ \texttt{|}\ \texttt{right}\ \boldsymbol{y}\ \texttt{=>}\ \boldsymbol{N_2}\ \texttt{end} \end{array}$$

計算規則:

$$\texttt{match left } \boldsymbol{M_1} \texttt{ with left } \boldsymbol{x} \texttt{ => } \boldsymbol{N_1}[\boldsymbol{x}] \texttt{ | right } \boldsymbol{y} \texttt{ => } \boldsymbol{N_2} \texttt{ end} \longrightarrow \boldsymbol{N_1}[\boldsymbol{M_1}]$$
$$\texttt{match right } \boldsymbol{M_2} \texttt{ with left } \boldsymbol{x} \texttt{ => } \boldsymbol{N_1} \texttt{ | right } \boldsymbol{y} \texttt{ => } \boldsymbol{N_2}[\boldsymbol{y}] \texttt{ end} \longrightarrow \boldsymbol{N_2}[\boldsymbol{M_2}]$$

型付け規則:

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

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

$$\frac{\boldsymbol{\Gamma} \vdash \boldsymbol{M} : \boldsymbol{S_1} + \boldsymbol{S_2} \qquad \boldsymbol{\Gamma},\ \boldsymbol{x} : \boldsymbol{S_1} \vdash \boldsymbol{N_1} : \boldsymbol{T} \qquad \boldsymbol{\Gamma},\ \boldsymbol{y} : \boldsymbol{S_2} \vdash \boldsymbol{N_2} : \boldsymbol{T} \qquad (\boldsymbol{x}, \boldsymbol{y} \notin \mathit{dom}(\boldsymbol{\Gamma}))}{\boldsymbol{\Gamma} \vdash \texttt{match } \boldsymbol{M} \texttt{ with left } \boldsymbol{x} \texttt{ => } \boldsymbol{N_1} \texttt{ | right } \boldsymbol{y} \texttt{ => } \boldsymbol{N_2} \texttt{ end} : \boldsymbol{T}} \quad \text{(T-SMatch)}$$

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)

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，

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots \\ \boldsymbol{\Gamma} \vdash \mathtt{A} & \boldsymbol{\Gamma} \vdash \mathtt{B}\end{array}}{\boldsymbol{\Gamma} \vdash \mathtt{A} * \mathtt{B}}\ \wedge\text{-I}}{\boldsymbol{\Gamma} \vdash \mathtt{B}}\ \wedge\text{-E2}$$

という導出を考える．この最終的な結論は $\boldsymbol{\Gamma} \vdash \mathtt{B}$ だが，それは既に上で得られているものであり，最後の $\wedge$-I, $\wedge$-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

$$\begin{array}{c}\vdots \\ \boldsymbol{\Gamma} \vdash \mathtt{B}\end{array}$$

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

$$\dfrac{\dfrac{\vdots \quad \vdots}{\Gamma \vdash M_1 : \mathtt{A} \quad \Gamma \vdash M_2 : \mathtt{B}}\ \text{T-Pair}}{\dfrac{\Gamma \vdash (M_1, M_2) : \mathtt{A} * \mathtt{B}}{\Gamma \vdash \mathtt{snd}\ (M_1, M_2) : \mathtt{B}}\ \text{T-Snd}} \longrightarrow \Gamma \vdash \vdots\ M_2 : \mathtt{B}$$

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，

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

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

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma, x : S, \cdots \vdash x : S} \quad \cdots \quad \overline{\Gamma, x : S, \cdots \vdash x : S}\ \text{T-Var}}{\Gamma, x : S \vdash \vdots\ M[x] : T}}{\Gamma \vdash \mathtt{fun}\ x : S \Rightarrow M[x] : S \to T}\ \text{T-Abs} \qquad \Gamma \vdash \vdots\ N : S}{\Gamma \vdash (\mathtt{fun}\ x : S \Rightarrow M[x])\ N : T}\ \text{T-App}$$

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．

$$\dfrac{\vdots \quad \cdots \quad \vdots}{\Gamma, \cdots \vdash N : S \quad \cdots \quad \Gamma, \cdots \vdash N : S}$$
$$\Gamma \vdash M[N] : T$$

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，

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash M : S_1}\ \text{T-Left}}{\Gamma \vdash \mathtt{left}\ M : S_1 + S_2} \quad \dfrac{\overline{\Gamma, x : S, \cdots \vdash x : S} \quad \cdots \quad \overline{\Gamma, x : S, \cdots \vdash x : S}\ \text{T-Var}}{\Gamma, x : S_1 \vdash \vdots\ N_1[x] : T} \qquad \Gamma, y : S_2 \vdash \vdots\ N_2[y] : T}{\Gamma \vdash \mathtt{match\ left}\ M\ \mathtt{with\ left}\ x \Rightarrow N_1[x]\ \mid\ \mathtt{right}\ y \Rightarrow N_2\ \mathtt{end} : T}\ \text{T-SMatch}$$

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

$$\dfrac{\dfrac{\vdots}{\boldsymbol{\Gamma}, \cdots \vdash \boldsymbol{M} : \boldsymbol{S_1}} \quad \cdots \quad \dfrac{\vdots}{\boldsymbol{\Gamma}, \cdots \vdash \boldsymbol{M} : \boldsymbol{S_1}}}{\boldsymbol{\Gamma} \vdash \boldsymbol{N_1}[\boldsymbol{M}] : \boldsymbol{T}}$$

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．

5　全称量化子と依存関数型

最後に全称量化子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．

全称量化子 $\forall \boldsymbol{x} : \boldsymbol{T}, \boldsymbol{P}$ 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，全称量化子 $\forall \boldsymbol{x} : \boldsymbol{T}, \boldsymbol{P}[\boldsymbol{x}]$ 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「$\boldsymbol{T}$ 型の値 $\boldsymbol{M}$ を受け取って $\boldsymbol{P}[\boldsymbol{M}]$ の証明を返す関数」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．返り値の型はどうなるだろうか．n を受け取った時の返り値は

$$\underbrace{\texttt{cons 0 (cons 0 ... (cons 0}}_{n}\ \texttt{nil)...)}$$

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返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

$$\Pi \boldsymbol{x} : \boldsymbol{S}.\ \boldsymbol{T}[\boldsymbol{x}]$$

という形をしており，引数 $\boldsymbol{x}$ の型が $\boldsymbol{S}$ で返り値の型が $\boldsymbol{T}$ である (ただし，$\boldsymbol{T}$ には引数 $\boldsymbol{x}$ が現れる可能性がある) ことを表している．例えば，`makezero` の型は

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

となる．

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

```
makezero 3 : natlist3
makezero (n + m) : natlist(n + m)
```

などとなる．

単純な関数型 $\boldsymbol{S}$`->`$\boldsymbol{T}$ は，実は依存関数型のうち，引数が返り値の型に現れない (返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，Π`n : nat. 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．