

# Coq の論理

## 1 命題論理

**論理式** 論理式は以下の結合子から定義される.

$A, B ::=$	$True \mid False$	定数
	$  X$	論理変数
	$  A \supset B$	含意
	$  A \wedge B$	論理積
	$  A \vee B$	論理和

否定はないが, 便宜のために  $\neg A = A \supset False$  とおく.

**導出規則** 自然演繹体系では真の論理式は以下の規則より導出される.  
 $\Delta$  を論理式の集合とする.  $True$  は常に  $\Delta$  に含まれる.

<b>公理</b>	$\Delta \vdash A$ ( $A$ は $\Delta$ に含まれる)	<b><math>\wedge</math> 導入</b>	$\frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \wedge B}$
<b><math>\supset</math> 導入</b>	$\frac{\Delta, A \vdash B}{\Delta \vdash A \supset B}$	<b><math>\wedge</math> 除去</b>	$\frac{\Delta \vdash A \wedge B}{\Delta \vdash A} \quad \frac{\Delta \vdash A \wedge B}{\Delta \vdash B}$
<b><math>\supset</math> 除去</b>	$\frac{\Delta \vdash A \quad \Delta \vdash A \supset B}{\Delta \vdash B}$	<b><math>\vee</math> 導入</b>	$\frac{\Delta \vdash A}{\Delta \vdash A \vee B} \quad \frac{\Delta \vdash B}{\Delta \vdash A \vee B}$
<b>背理法</b>	$\frac{\Delta, \neg A \vdash False}{\Delta \vdash A}$	<b><math>\vee</math> 除去</b>	$\frac{\Delta \vdash A \vee B \quad \Delta, A \vdash C \quad \Delta, B \vdash C}{\Delta \vdash C}$

**恒真式** 命題論理の恒真式は  $True$  だけを仮定して導出できる式である.

例えば,  $P \supset P \wedge P$  や  $P \supset (P \supset Q) \supset Q$  は恒真式である. それぞれの導出を以下に示す.

$$\frac{\frac{P \vdash P \quad P \vdash P \text{ (公理)}}{P \vdash P \wedge P} \text{ (}\wedge\text{ 導入)}}{\vdash P \supset P \wedge P} \text{ (}\supset\text{ 導入)}$$

$$\frac{\frac{P, P \supset Q \vdash P \quad P, P \supset Q \vdash P \supset Q \text{ (公理)}}{P, P \supset Q \vdash Q} \text{ (}\supset\text{ 導入)}}{\vdash P \supset (P \supset Q) \supset Q} \text{ (}\supset\text{ 除去)}$$

## 2 命題と型の対応

カリヤ・ハワード同型により, 命題論理と型理論 (型付  $\lambda$  計算) が対応している.  
 具体的には, 以下のような対応が見られる.

命題 (論理式)	型
証明 (導出)	プログラム
仮定 $\Gamma$	型環境 $\Gamma$
$\supset$	$\rightarrow$
$\wedge$	$*$

導出規則と型付け規則も基本的には1対1で対応している。それぞれの体系を少し修正すると以下の定理がなりたつ。

**定理 1 (Curry-Howard 同型)** ある同型  $\langle \_ \rangle : \text{命題} \rightarrow \text{型}$  が存在し、任意の  $\Delta$  と  $A$  について、導出  $\Pi$  より  $\Delta \vdash A$  が示せるならば、 $\Pi$  からプログラム  $M$  が作れ、 $\langle \Delta \rangle \vdash M : \langle A \rangle$ 。また、任意の  $\Gamma, M, \tau$  について型理論で  $\Gamma \vdash M : \tau$  が導出できれば、命題論理において  $\langle \Gamma \rangle^{-1} \vdash \langle \tau \rangle^{-1}$  が導出できる。

修正の内容は二種類ある。

まず、前回の**不動点**の規則は矛盾を生んでしまう。具体的には、 $\theta = \text{True}$  と  $\tau = \text{False}$  にすると、以下の導出が可能になる。

$$\frac{\Gamma, f : \text{True} \rightarrow \text{False}, x : \text{True} \vdash f x : \text{False}}{\Gamma \vdash \text{fix } f (x:\theta) := f x : \text{True} \rightarrow \text{False}}$$

しかし、Coq の本当の**不動点**の規則はさらに  $f$  が  $x$  より小さな引数に適用されることを求めているので、この矛盾が実際には起きない。本当の規則が複雑なのでここには書かない。

もう一つは、**背理法**に対する規則は Coq の型体系にはない。それは Coq は通常の命題論理 (**古典論理**) に基いているのではなく、それと少し異なる**直感主義論理**に基いているからである。もしも命題論理を直感主義にするならば、背理法を以下の**矛盾**という規則に置き換えればいい。

$$\text{矛盾} \quad \frac{\Delta \vdash \text{False}}{\Delta \vdash A}$$

要するに、矛盾 (*False*) が証明できれば、何でも証明できるようにする。古典論理では背理法よりそれが導出できるが、背理法のない直観主義論理ではこの新しい規則が必要になる。

Coq の論理はこの直観主義論理とちょうど一致する。メリットとして、全ての証明が計算的な意味を持つ—証明は関数である。

しかし、逆に Coq の中で古典論理の証明をしたいときもある。ほとんどの定理は背理法なしで証明できるものの、証明できない定理もある上、単に背理法が便利なきもある。そのとき、Coq の論理に新しい公理として以下の規則を導入すればいい。

$$\neg\neg\text{除去} \quad \Delta \vdash \neg\neg A \supset A$$

この公理と抽象を組合せると背理法が導出可能になる。

### 3 Coq で定理の証明

前述の Curry-Howard 同型のおかげで、Coq の中で直接に命題を書くことができる。その型を満すプログラムが見つかれば、定理になる。

**変数宣言** まずは、準備として論理変数の宣言を行う。Section というコマンドを使うと、局所的な論理変数が宣言できるようになる。宣言自体は Variables コマンドを使う。そして、宣言範囲が終ると End コマンドでセクションを閉じる。

```
Section Koushin.
```

```
Variables P Q : Prop.
P is assumed
Q is assumed
```

論理式自身は型であると先に説明したが、通常の型の型だった Set と異なり、論理式の型は Prop になる。普段はあまり影響はないが、区別すると便利なことができる。

## 命題と証明プログラム

まず、前の二つの恒真式を証明してみよう。  
2つ目は関数適用だけなので、簡単にできる。

```
Theorem modus_ponens : P -> (P -> Q) -> Q.      (* 名前を付けなければならない *)
Proof (fun p pq => pq p).
modus_ponens is defined
```

```
Print modus_ponens.                               (* 実際には関数定義と変わらない *)
modus_ponens = fun (p : P) (pq : P -> Q) => pq p
              : P -> (P -> Q) -> Q
```

しかし、一つ目ではデータの直積ではなく、命題の論理積を使ったので、作り方を調べなければならぬ。

```
SearchPattern ( _ /\ _ ).                         (* 論理積を返す関数(定理)を調べる *)
andb_prop: forall a b : bool, (a && b)%bool = true -> a = true /\ b = true
conj: forall A B : Prop, A -> B -> A /\ B
iff_and: forall A B : Prop, (A <-> B) -> (A -> B) /\ (B -> A)
```

この中では、`conj` が期待の操作をしている。

```
Theorem and_self : P -> P /\ P.
Proof (fun x => conj x x).
and_self is defined
```

## 作戦 (tactic) の利用

上のように、プログラムを与えることで定理を証明することができる。しかし、複雑な定理になると、途中で出て来る命題が煩雑になり、正しいプログラムを書くのが至難の技になる。

通常は、定理は関数と違う定義方法を使う。証明モードに入り、作戦 (tactic) によって証明を構築していく。各 tactic は導出規則と対応している。

```
Theorem modus_ponens' : P -> (P -> Q) -> Q.      (* 異なる名前にする *)
1 subgoal                                       (* 証明の状況が表示される *)
```

```
P : Prop
Q : Prop
=====
P -> (P -> Q) -> Q
```

```
Proof.
intros p pq.                                   (* 仮定に名前を付ける (抽象) *)
```

```
p : P
pq : P -> Q
=====
Q
```

```
apply pq.                                     (* 目標を関数 pq の結果とみなす (適用) *)
```

```
p : P
pq : P -> Q
=====
P
```

```
assumption.
```

```

Proof completed.
Qed.
modus_ponens' is defined

```

実際の証明をもう一度みよう.

```

Theorem modus_ponens' : P -> (P -> Q) -> Q.
Proof.
  intros p pq.
  apply pq.
  assumption.
Qed.

```

and\_self について同じことをする.

```

Theorem and_self' : P -> P /\ P.
Proof.
  intros p.
  1 subgoal

```

```

  p : P
  =====
  P /\ P

```

```

  split.
  2 subgoals

```

(\* 論理積の導入 (∧ 導入) \*)  
 (\* 前提が二つある \*)

```

  p : P
  =====
  P

```

```

subgoal 2 is:
P

```

```

  assumption.
  1 subgoal

```

(\* 順番に解いていく \*)

```

  p : P
  =====
  P

```

```

  assumption.
Qed.
and_self' is defined

```

```

Print and_self'.
and_self' = fun p : P => conj p p
            : P -> P /\ P

```

(\* 実際の定義は前と変わらない \*)

## セクションを閉じる

```

End Koushin.

```

```

Print and_self.
and_self =
fun (P : Prop) (x : P) => conj x x
  : forall P : Prop, P -> P /\ P

```

(\* 必要な変数が定義に挿入される \*)

## 否定に関する定理

証明状態の表示が作戦を読みにくくするので、これ以降は省くことにする。自分で Coq の中で実行して、確認して下さい。

```
Section Negation.
```

```
Variables P Q : Prop.
```

```
Theorem DeMorgan : ~ (P ∨ Q) -> ~ P ∧ ~ Q.
```

```
Proof.
```

```
  intros npq.
```

```
  split; intros q. (* ; で両方の subgoal について intros q を行う *)
```

```
    apply npq.
```

```
    left.
```

```
    (* ∨ 導入の左を使う *)
```

```
    assumption.
```

```
  elim npq.
```

```
  right.
```

```
  assumption.
```

```
Qed.
```

```
DeMorgan is defined
```

しかし、双対的な定理  $(\neg(P \wedge Q) \supset \neg P \vee \neg Q)$  は直観主義論理ではなりたたない。Hypothesis コマンドによって二重否定の除去を仮定すると証明できる。ちなみに、Hypothesis コマンドは Variables の異名でしかなくて、動作は全く同じである。

```
Hypothesis classic : forall P, ~~P -> P.
```

```
(* 任意の P について *)
```

```
classic is assumed
```

```
Theorem DeMorgan' : ~ (P ∧ Q) -> ~ P ∨ ~ Q.
```

```
Proof.
```

```
  intros npq.
```

```
  apply classic.
```

```
  intro nnpq.
```

```
  apply npq.
```

```
  clear npq.
```

```
(* 不要な仮定を忘れる *)
```

```
  split; apply classic.
```

```
    intros np.
```

```
    apply nnpq.
```

```
    left.
```

```
    assumption.
```

```
  intros np; apply nnpq; right; assumption.
```

```
Qed.
```

```
DeMorgan' is defined
```

```
End Negation.
```

## 仮定を破壊する

Coq の帰納的データ型に対して、値を破壊しながら中身を取り出すという tactic が便利である。直接に対応する論理規則はないが、当然ながら他の論理規則から同じ結果を導くことは可能である。

```
Section Destruct.
```

```
Variables P Q : Prop.
```

```
Theorem and_comm : P ∧ Q -> Q ∧ P.
```

```

Proof.
  intros pq.
  destruct pq as [p q].
  split; assumption.
Qed.
and_comm is defined

```

(\* 中身を取り出す \*)  
(\* 一気に終わらせる \*)

```

Theorem or_comm : P ∨ Q -> Q ∨ P.
Proof.
  intros pq.
  destruct pq as [p | q].
  right; assumption.
  left; assumption.
Qed.
or_comm is defined

```

(\* 場合が二つある \*)

End Destruct.

### 論理規則と tactic の対応

論理規則	型付け規則	作戦
公理	変数	assumption
⊃ 導入	抽象	intros <i>h</i>
⊃ 除去	適用	apply <i>h</i>
矛盾		elimtype False
∧ 導入	直積	split
∧ 除去		destruct <i>h</i> as [ <i>h</i> <sub>1</sub> <i>h</i> <sub>2</sub> ]
∨ 導入	直和	left, right
∨ 除去	match	destruct <i>h</i> as [ <i>h</i> <sub>1</sub>   <i>h</i> <sub>2</sub> ]

**練習問題 3.1** 以下の定理を *Coq* で証明せよ。

```

Section Coq2.
Variables P Q R : Prop.
Theorem imp_trans : (P -> Q) -> (Q -> R) -> P -> R.
Theorem not_false : ~False.
Theorem double_neg : P -> ~~P.
Theorem contraposition : (P -> Q) -> ~Q -> ~P.
Theorem and_assoc : P ∧ (Q ∧ R) -> (P ∧ Q) ∧ R.
Theorem and_distr : P ∧ (Q ∨ R) -> (P ∧ Q) ∨ (P ∧ R).
Theorem absurd : P -> ~P -> Q.
End Coq2.

```