

述語論理と SSReflect のタクティック

1 述語論理

前回見た命題論理は、推論の概念を捉えているが、具体的な対象に対して議論することができない。我々が一般的に使う論理はその拡張である述語論理になる。最も見慣れているのは一階述語論理だが、実数の形式化などには二階述語論理が必要。

論理式 二階述語論理の論理式は以下の結合子から定義される。(X を除くと一階述語論理に限定される)

$t ::= x$	項変数	$P, Q ::= \dots$	命題
c	項定数	ϕ	命題述語
$f(t_1, \dots, t_n)$	項関数	$\phi(t_1, \dots, t_n)$	述語
$\phi ::= p$	述語名	$\forall v.P$	全称
X	述語変数	$\exists v.P$	存在
$v ::= x \mid X$	変数	$t_1 = t_2$	等価性
$\sigma ::= t \mid \phi$	代入		

導出規則 命題論理の導出規則に以下の規則を加える。

\forall 導入	$\frac{\Delta \vdash P \quad v \notin fv(\Delta)}{\Delta \vdash \forall v.P}$	\exists 導入	$\frac{\Delta \vdash [t/x]P \quad \Delta \vdash [\phi/X]P}{\Delta \vdash \exists x.P \quad \Delta \vdash \exists X.P}$
\forall 除去	$\frac{\Delta \vdash \forall x.P \quad \Delta \vdash \forall X.P}{\Delta \vdash [t/x]P \quad \Delta \vdash [\phi/X]P}$	\exists 除去	$\frac{\Delta \vdash \exists v.P \quad \Delta, P \vdash Q \quad v \notin fv(\Delta, Q)}{\Delta \vdash Q}$
反射率	$\Delta \vdash t = t$	代入	$\frac{\Delta \vdash [t_1/x]P \quad \Delta \vdash t_1 = t_2}{\Delta \vdash [t_2/x]P}$

ここで自由変数 $fv(P)$ と代入が利用される。

$$\begin{aligned}
 fv(x) &= \{x\} & fv(c) &= \emptyset & fv(f(t_1, \dots, t_n)) &= \bigcup fv(t_i) \\
 fv(True) &= fv(False) = fv(p) = \emptyset & & & fv(\phi(t_1, \dots, t_n)) &= fv(\phi) \cup \bigcup fv(t_i) \\
 fv(P \supset Q) &= fv(P \wedge Q) = fv(P \vee Q) = fv(P) \cup fv(Q) & & & fv(X) &= \{X\} \\
 fv(\forall x.P) &= fv(\exists x.P) = fv(P) \setminus \{x\} & & & &
 \end{aligned}$$

$$\begin{aligned}
 [t/x]x &= t & [t/x]y &= y & [\phi/X]t &= t \\
 [t/x]c &= c & [t/x]f(t_1, \dots, t_n) &= f([t/x]t_1, \dots, [t/x]t_n) & &
 \end{aligned}$$

$$\begin{aligned}
 [\sigma/v]True &= True & [\sigma/v]False &= False & [\phi/X]X &= \phi & [\sigma/v]X &= X \\
 [\sigma/v](P \supset Q) &= [\sigma/v]P \supset [\sigma/v]Q & & & & & \wedge, \vee \text{も同様} \\
 [\sigma/v](\phi(t_1, \dots, t_n)) &= [\sigma/v]\phi([\sigma/v]t_1, \dots, [\sigma/v]t_n) & & & & & \\
 [\sigma/v]\forall w.P &= \forall w.[\sigma/v]P \quad (w \notin fv(\sigma) \cup \{v\}) & & & [\sigma/v]\forall v.P &= \forall v.P \\
 [\sigma/v]\exists w.P &= \exists w.[\sigma/v]P \quad (w \notin fv(\sigma) \cup \{v\}) & & & [\sigma/v]\exists v.P &= \exists v.P
 \end{aligned}$$

導出の例

$$\frac{\frac{\forall x.\text{human}(x) \supset \text{mortal}(x) \vdash \forall x.\text{human}(x) \supset \text{mortal}(x)}{\forall x.\text{human}(x) \supset \text{mortal}(x) \vdash \text{human}(S) \supset \text{mortal}(S)} \quad \text{human}(S) \vdash \text{human}(S)}{\forall x.\text{human}(x) \supset \text{mortal}(x), \text{human}(S) \vdash \text{mortal}(S)}$$

2 Coqとの対応

Coqでは項と述語が型をもっており、代入が型を保たなければならない。さらに、型自身が型(ソート)をもっている。

$\vdash t, x : T$ かつ $\vdash T : \text{Set}$ または $\vdash T : \text{Type}$

$\vdash P : \text{Prop}$ $\vdash p, X : T_1 \rightarrow \dots \rightarrow T_n \rightarrow \text{Prop}$

さらに、全ての $T \rightarrow P$ が $\forall x : T, P$ の略記法であり、 \forall と \rightarrow に関する規則が統一されている。

抽象	$\frac{\Delta, v : S \vdash t : T}{\Delta \vdash (\text{fun } v : S \Rightarrow t) : \forall v : S, T}$	適用	$\frac{\Delta \vdash t : \forall v : S, T \quad \Delta \vdash u : S}{\Delta \vdash (t \ u) : [u/v]T}$

$\Delta \vdash t : T$ が項 t が仮定 Δ の元で型 T をもつという意味である。 T が命題なら ($\vdash T : \text{Prop}$), t はその証明項。今までの導出規則が拡張され、 \vdash と結論の間に証明項が含まれる。ただし、抽象が \rightarrow の導入と \forall の導入を兼ね、適用が \rightarrow の除去と \forall の除去と述語および項関数の構成を兼ねる。たとえば、公理の規則は $\Delta \vdash x : T$ ($x : T \in \Delta$) になる。

上の導出の例は適用2回でできる。

```
Section Socrates.
Variable A : Set.
Variables human mortal : A -> Prop.
Variable socrates : A.

Hypothesis hm : forall x, human x -> mortal x.
Hypothesis hs : human socrates.

Theorem ms : mortal socrates.
Proof.
  apply: (hm socrates).
  assumption.
Qed.

Print ms.
ms = hm socrates hs
      : mortal socrates
End Socrates.
```

(* 定義を表示する *)
(* ((hm socrates) hs) *)

3 等式変換による証明

代入規則に対するタクティックは `rewrite` である。それを使うことで等式理論での証明が可能になる。

等価性の性質

```
Require Import ssreflect.
Section Eq.
  Variable T : Type.

  Lemma symmetry : forall x y : T, x = y -> y = x.
  Proof.
    move=> x y exy.
    rewrite exy. (* x を y に書き換える *)
    done. (* 反射率で終わらせる *)
  Restart.
  by move=> x y ->. (* => の後ろなら -> で書き換えられる *)
  Qed.

  Lemma transitivity : forall x y z : T, x = y -> y = z -> x = z.
  Abort.
End Eq.
```

実はこの2つの補題(および reflexivity)はタクティックとして提供されている。

群の公理化

```
Section Group.
  Variable G : Set.
  Variable e : G.
  Variable op : G -> G -> G.
  Notation "a * b" := (op a b). (* op を * と書けるようにする *)
  Variable inv : G -> G.
  Hypothesis associativity : forall a b c, (a * b) * c = a * (b * c).
  Hypothesis left_identity : forall a, e * a = a.
  Hypothesis right_identity : forall a, a * e = a.
  Hypothesis left_inverse : forall a, inv a * a = e.
  Hypothesis right_inverse : forall a, a * inv a = e.

  Lemma unit_unique : forall e', (forall a, a * e' = a) -> e' = e.
  Proof.
    move=> e' He'.
    rewrite -[RHS]He'. (* 右辺を書き換える *)
    rewrite (left_identity e'). (* 公理をの引数を指定する *)
    done.
  Qed.

  Lemma inv_unique : forall a b, a * b = e -> a = inv b.
  Proof.
    move=> a b abe.
    have : a * b * inv b = e * inv b.
      by rewrite abe.
    rewrite associativity right_inverse left_identity right_identity.
    done. (* 書き換えはまとめて書ける *)
  Qed.

  Lemma inv_involutive : forall a, inv (inv a) = a.
  Abort.
End Group.
Check unit_unique.
```

練習問題 3.1 transitivity と inv_involutive を証明に変えよ。

4 全称と存在

\forall と \exists の間に De Morgan の法則がなりたつ。前回と同様に、 \exists を導出しようとしたときに `classic` を使わなければならない。

Section Laws.

Variables (A:Set) (P Q : A->Prop).

Lemma DeMorgan2 : (\sim exists x, P x) -> forall x, \sim P x.

Proof.

move=> N x Px. elim: N. by exists x.

Qed.

Theorem exists_or :

(exists x, P x \vee Q x) -> (exists x, P x) \vee (exists x, Q x).

Proof.

move=> [x [Px | Qx]]; [left|right]; by exists x.

Qed.

Hypothesis EM : forall P, P \vee \sim P.

Lemma DeMorgan2' : \sim (forall x, P x) -> exists x, \sim P x.

Proof.

move=> nap.

case: (EM (exists x, \sim P x)) => //. (* 背理法 *)

move=> nnp.

elim: nap => x. (* (forall x, P x) を証明する *)

case: (EM (P x)) => //. (* 背理法 *)

move=> npx.

elim: nnp.

by exists x.

Qed.

End Laws.

練習問題 4.1 以下の定理を *Coq* で証明せよ。

Require Import ssreflect.

Section Coq3.

Variable A : Set.

Variable R : A -> A -> Prop.

Variables P Q : A -> Prop.

Theorem exists_postpone :

(exists x, forall y, R x y) -> (forall y, exists x, R x y).

Theorem exists_mp : (forall x, P x -> Q x) -> ex P -> ex Q.

Theorem or_exists :

(exists x, P x) \vee (exists x, Q x) -> exists x, P x \vee Q x.

Hypothesis EM : forall P, P \vee \sim P. (* 残りは排中律を使う *)

Variables InPub Drinker : A -> Prop.

Theorem drinkers_paradox :

(exists consumer, InPub consumer) -> (* パブには客がいる *)

exists man, InPub man \wedge (* 彼が飲むと全員が読むような客 がいる *)

(Drinker man -> forall other, InPub other -> Drinker other).

```
Theorem remove_c : forall a,  
  (forall x y, Q x -> Q y) ->  
  (forall c, ((exists x, P x) -> P c) -> Q c) -> Q a.  
End Coq3.
```

5 タクティク・タクティカル・修飾子

前回はタクティクと論理規則との関係について述べたが、SSREFLECTのタクティクが多くの機能を秘めており、ここでその一部を説明する。

基本タクティク

move, apply, done, case, split, left, right, elim, have, suff, rewrite, set でほぼ全ての証明ができるが、特に move と rewrite が修飾子を多く使う。

CoQの証明状態を以下とする。(ただし焦点がない場合もある)

```
假定 Δ
=====
焦点 P -> 結論 Q
```

各タクティクがゴールの各部 ($\Delta; P \vdash Q$) を論理的規則に基づいて変えていく。その変化を説明する。

move 修飾子なしでは何もしないが、修飾子によって假定と前提の移動や左規則の適用が可能。

apply ゴールを適用した定理の前提に変える。規則 Apply-R を参照。

前の状態: $\Delta; P_1 \rightarrow \dots \rightarrow P_n \vdash Q$

後の状態: $\Delta; \vdash P_i$ ($1 \leq i \leq n$) に置き換わる

done 様々な自明な解決法を試み、解決できなければエラーを起こす。公理、矛盾、反射率を含む。

case 焦点に対して場合分けを行う。単独では `move=> []` とほぼ同じ。規則 \wedge -L と \vee -L を参照。

split \wedge -R を参照。ゴールを二つに分割する。

left, right \vee -R を参照。ゴールの左か右を選ぶ。間違えると証明ができない可能性が大きい。

elim 焦点の返り値 (結論) に対して場合分けを行う。型によって帰納法の原理が適用される。併せて、apply と同様に焦点の前提もゴールのリストに加えられる。焦点が否定ならば

前の状態: $\Delta; \neg P \vdash Q$

後の状態: $\Delta; \vdash P$

have $H : P$

新しい假定 $H : P$ を証明した上で現在のゴールの証明を続ける。

前の状態: $\Delta; \vdash Q$

後の状態: $\Delta; \vdash P$ と $\Delta, H : P; \vdash Q$

假定名 H が省略されると代わりに $\Delta; P \vdash Q$ になる。

suff $H : P$

have と同じだが、ゴールの順番が逆になる。新しい假定の元で現在のゴールを証明した後、その假定を証明しなければならない。have $H : P$; last first とほぼ同じ。

rewrite /def

ゴールの焦点および結論の中で定義 *def* を展開する。詳しい rewrite の修飾子は後で述べる。

rewrite lemma

補題 *lemma* の結論が等式 $t_1 = t_2$ ならば、ゴールの焦点および結論の中で t_1 を t_2 に書き換える。等式の変数が自動的に選ばれる。*lemma* の前提がゴールリストに追加される。

set $x := t$

假定に定義 $x := t : T$ を加える。 x は假定名、 t は項、 T は t の型。同時に結論の中に t を x に置き換える (rewrite $-/x$ と同じ)。

t の中に穴「 $_$ 」を含めてもいい。その場合、結論の中に当て嵌る項を探し、その穴を自動的に埋める。

have $H := t$

H は假定名、 t は項。 Δ の元で t の型が T なら、假定 $H : T$ を置く。 t が証明項なら、 T が命題になる。

基本タクティカル タクティックの前後に書き、そのタクティックの動作を拡張する。

: move, apply, case, elim の直後に使える。タクティック本体を実行する前に前提を置く。

前の状態: $\Delta; \vdash Q$

タクティカル: $: H_1 \dots H_n$

後の状態: $\Delta'; P_1 \vdash P_2 \rightarrow \dots \rightarrow P_n \rightarrow Q$

$H_i : P_i$ が Δ に含まれる假定なら、 Δ' から除かれる。そうでなければ、 H_i は Δ の元で片付け可能な項で、 P_i はその型である。なお $H_i : P_i$ が Δ に含まれても、 (H_i) と書くことで置いたままにできる。また、不要になった假定 K があれば、 $\{K\}$ と書くことで消すことができる。

=> 全てのタクティックの後に使える。後に書かれた名前が順番に前提を焦点から假定に変える。

[] は焦点に左規則や分解規則を適用する。中に名前や修飾子を書いてもいい。 $[l_1 \mid \dots \mid l_n]$ の形で分解後の各場合に対する列も書ける。

- move 以外のタクティックと組合せたとき、=>の後の最初の [] か $[l_1 \mid \dots \mid l_n]$ が生成された各ゴールに対するものになり、分解を行わない。その前に-を書くとは分解になる。

/= は焦点と結論を単純化する。

// は全てのゴールに対して done で解決を失敗せずに試みる。

//= はその両方を行う。

/H は項 H を焦点に適用する。

/($_$ H) は焦点を H 適用する。

-> 焦点が等式 $t_1 = t_2$ ならば t_1 を t_2 に書き換え、その後焦点が消される。

<- は逆向きに書き換える。

{ K } は「:」のときと同じ意味。

/H apply の直後に使うと H が結論に適用される。move や case の直後だと、タクティックを実行する前に H が焦点に適用される。

in $H_1 \dots H_n$ ほぼ: $H_1 \dots H_n => H_1 \dots H_n$ と同じ意味だが、操作が元の結論に適用されない。

tac₁; tac₂ tac₁ を実行した後に、生成された全てのゴールに対して tac₂ を実行する。

tac; [tac₁ | ... | tac_n]

tac が n 個のゴールを生成した場合、それぞれに対して tac₁, ..., tac_n を実行する。

by tactic

tactic; done と同じ。なお、tactic が「;」を含んでもいい。(次の「.」まで行く)

by [] done と同じ。

do n tactic tactic を n 回繰り返す。

do !tactic tactic を可能な限り繰り返す。

rewrite

rewrite は独自の構文を使う。基本的には、定理の名前や適用を並べる。

```
rewrite lem1 lem2 (lem3 n 1)
```

各定理に対して、繰り返しや適用箇所の指定もできる。

`!lem` 定理 `lem` による書き換えを可能な限り繰り返す。

`n!lem` 定理 `lem` による書き換えを `n` 回繰り返す。

`?lem` 定理 `lem` を 0 または一回使う。

`-lem` 定理 `lem` を左向きに使う。

`{n}lem` `n` 番目の出現を書き換える

`[p]lem` パターン `p` (穴 “`_`” 含める項) にマッチする最初の出現を書き換える。

`[in p]lem` パターン `p` にマッチする最初の出現の中で書き換える。

`[X in p]lem` パターン `p` にマッチする最初最初の出現の中の `X` に当る部分を書き換える。

`[LHS]`, `[RHS]` それぞれ `[X in X = _]` と `[X in _ = X]` と同じ。

`/def` 定義 `def` を展開する。

`-/def` 定義 `def` を畳み込む。

`(lem1, ..., lemn)`

`lem1...lemn` を順番に試して、最初に成功してものを使う。他の修飾子 (特に「!」) と組合せてもいい。

`(_ : t1 = t2)`

`t1` を `t2` に置き換える。 `t1 = t2` がゴールリストに追加される。

上記の書き換え修飾子を組合せることができるが、順番に気を付けなければならない。

```
rewrite -{2}[_ + n]lem
```

また、定理や定義の間に評価修飾子 (`/=`, `//`, `//=`) を挿入してもいい。