Jacques Garrigue, 2021 年 6 月 28 日

計算機が支援する数学の証明

この集中講義では Coq および MathComp の理論と実践を見て行く．講義のサポートページは

http://www.math.nagoya-u.ac.jp/~garrigue/lecture/2021_kyushu/

1 Coq の論理

Coq は Calculus of Inductive Constructions (CIC) という型理論に基いている．型理論は型付きλ計算という計算体系と直観主義論理の証明論の間の同型 (カリー・ハワード対応) に基いた論理体系である．CIC は特に表現力が高い (一階述語論理や高階論理を含んでいる) が，まずその命題論理の部分から見ていくことにする．

1.1 直観主義の命題論理

論理式　論理式は以下の結合子から定義される．

$$\begin{array}{rll} P, Q ::= & \mathit{True} \mid \mathit{False} & \text{定数} \\ \mid & A & \text{論理変数} \\ \mid & P \to Q & \text{含意} \\ \mid & P \wedge Q & \text{論理積} \\ \mid & P \vee Q & \text{論理和} \end{array}$$

否定はないが，便宜のために $\neg P = P \to \mathit{False}$ 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．
Δ を論理式の集合とする．True は常に Δ に含まれる．

公理　$\Delta \vdash P$　(P はΔに含まれる)

$\to$ **導入**　$\dfrac{\Delta, P \vdash Q}{\Delta \vdash P \to Q}$

$\to$ **除去**　$\dfrac{\Delta \vdash P \quad \Delta \vdash P \to Q}{\Delta \vdash Q}$

矛盾　$\dfrac{\Delta \vdash \mathit{False}}{\Delta \vdash P}$

$\wedge$ **導入**　$\dfrac{\Delta \vdash P \quad \Delta \vdash Q}{\Delta \vdash P \wedge Q}$

$\wedge$ **除去**　$\dfrac{\Delta \vdash P \wedge Q}{\Delta \vdash P} \quad \dfrac{\Delta \vdash P \wedge Q}{\Delta \vdash Q}$

$\vee$ **導入**　$\dfrac{\Delta \vdash P}{\Delta \vdash P \vee Q} \quad \dfrac{\Delta \vdash Q}{\Delta \vdash P \vee Q}$

$\vee$ **除去**　$\dfrac{\Delta \vdash P \vee Q \quad \Delta, P \vdash R \quad \Delta, Q \vdash R}{\Delta \vdash R}$

この導出規則から背理法や排中律が導けない．もしも古典論理の体系が欲しければ，**矛盾**の代わりに以下の同値な規則のいずれかを使う．

背理法　$\dfrac{\Delta, \neg P \vdash \mathit{False}}{\Delta \vdash P}$　　$\neg\neg$**除去**　$\Delta \vdash \neg\neg P \to P$　　**排中律**　$\Delta \vdash P \vee \neg P$

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

$$\dfrac{\dfrac{\dfrac{P \wedge Q \vdash P \wedge Q}{P \wedge Q \vdash Q}\,(\wedge\text{除去}) \quad \dfrac{P \wedge Q \vdash P \wedge Q}{P \wedge Q \vdash P}\,(\wedge\text{除去})}{P \wedge Q \vdash Q \wedge P}\,(\wedge\text{導入})}{\vdash P \wedge Q \to Q \wedge P}\,(\to\text{導入}) \qquad \dfrac{\dfrac{\dfrac{P, P \to Q \vdash P \quad P, P \to Q \vdash P \to Q}{P, P \to Q \vdash Q}\,(\to\text{除去})}{P \vdash (P \to Q) \to Q}\,(\to\text{導入})}{\vdash P \to (P \to Q) \to Q}\,(\to\text{導入})$$

(公理)　(公理)

1.2　Coq での証明

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．Coq の出力をイタリック体で表示している．

```
Section Koushin.

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

変数を宣言するときにその型も書かなければならないが，論理式の型は Prop になる．

命題と証明

まず，二つ目の恒真式を証明してみよう．定理を宣言すると証明モードに移る．

```
Theorem modus_ponens : P -> (P -> Q) -> Q.          (* 名前を付けなければならない *)
1 subgoal                                              (* 証明の状況が表示される *)
  P, Q : Prop
  ============================
   P -> (P -> Q) -> Q
```

線の上は Δ の中身，下は結論だが，今は Δ に変数の宣言しかない．P と Q が命題であるという宣言であり，P や Q が成り立つという仮定ではない．$\to$ **導入**にあたる Coq のタクティック (証明命令) は intros である．$\to$ **除去**は apply．**公理**は assumption.

```
Proof.
  intros p pq.                                      (* 仮定に名前を付ける（導入）*)
1 subgoal
  p : P
  pq : P -> Q
  ============================
   Q

  apply pq.                                         (* 目標を仮定 pq の帰結とみなす（除去）*)
1 subgoal
  p : P
  pq : P -> Q
  ============================
   P

  assumption.                                       (*（公理）*)
  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.
```

証明が長い訳ではないが，少し違和感がある．直感的に二つ目の前提を一つ目に適用すれば結論が出るはずなのに，逆にそれを結論に「適用」している．証明を結論から作っていくので，考え方が逆になってしまい，証明が長くなると追いにくい．

2 SSReflect での証明

上の自然演繹体系は Coq の中の論理によく合っているが，証明の構築には不便である．証明の構築は，証明したい命題と仮定を下に書き，証明木を延して行く形で行うが，自然演繹の場合では，規則は結論 (右側) に対するものばかりで，その結論を変形させる「逆算」で行うことになる．通常の数学の証明は仮定から性質を導く「前進」方向で行うことが多い．Gentzen のシークエント計算が結論を扱う「右規則」とは別に仮定を扱う「左規則」も持っている．ここでは，そういう左規則を自然演繹に追加する形で SSReflect の証明方法を反映した論理体系を作る．

2.1 SSReflect の論理

新しい判定の形は

$$\Delta; P \vdash Q$$

Δ には名前付きの仮定が入っている．P は焦点と言い，左 (L) 規則の対象となる．焦点は空白でもいい．Q は結論といい，右 (R) 規則の対象である．

Focus $\dfrac{\Delta; P \vdash Q}{\Delta; \vdash P \to Q}$

Unfocus $\dfrac{\Delta; \vdash P \to Q}{\Delta; P \vdash Q}$

Intro $\dfrac{\Delta, h : P; \vdash Q}{\Delta; P \vdash Q}$

Revert $\dfrac{\Delta; P \vdash Q}{\Delta, h : P; \vdash Q}$

Apply-R $\dfrac{\Delta; \vdash P_1 \quad \dots \quad \Delta; \vdash P_n}{\Delta; P_1 \to \dots \to P_n \to Q \vdash Q}$

Apply-L1 $\dfrac{\Delta; \vdash P \to Q \quad \Delta; Q \vdash R}{\Delta; P \vdash R}$

Apply-L2 $\dfrac{\Delta; \vdash P \quad \Delta; Q \vdash R}{\Delta; P \to Q \vdash R}$

公理 $\Delta; P \vdash P$

矛盾 $\Delta; \mathit{False} \vdash P$

カット $\dfrac{\Delta; \vdash P \quad \Delta; P \vdash Q}{\Delta; \vdash Q}$

∧-R $\dfrac{\Delta; \vdash P \quad \Delta; \vdash Q}{\Delta; \vdash P \wedge Q}$

∧-L $\dfrac{\Delta; P \vdash Q \to R}{\Delta; P \wedge Q \vdash R}$

∨-R $\dfrac{\Delta; \vdash P}{\Delta; \vdash P \vee Q} \quad \dfrac{\Delta; \vdash Q}{\Delta; \vdash P \vee Q}$

∨-L $\dfrac{\Delta; P \vdash R \quad \Delta; Q \vdash R}{\Delta; P \vee Q \vdash R}$

規則は下から上へと読む．**Focus** は結論の前提を焦点に移す．**Unfocus** は逆に焦点された命題を結論の前提に戻す．SSReflect ではこの二つの規則は暗に使われ，焦点と結論の前提が区別されない．**Intro** は焦点に名前を付けて仮定に移す．**Revert** は仮定を焦点に戻す．基本的にはこの 4 つの規則が仮定を動かしているいるだけだ．**Apply-R** は焦点を使って結論を証明するが，その際焦点の全ての前提を証明する必要が新たに生じる．**Apply-L1** は補題 $P \to Q$ を使って焦点を変える．**Apply-L2** は逆に補題 P を使って焦点の前提をみたす．この 3 つの規則を右側の**カット**で代用することができるので，論理的には必要ではな

いが，証明を自然に進めるのに使う．右側の規則は右規則である導入が自然演繹と変らないが，除去に当たる左規則やカットが焦点を利用している．特別な場合として，$\wedge$-**L** では焦点に P しか入らないので，Q が前提に戻される．

2.2 SSReflect のタクティック

SSReflect の論理は多くの規則をもっているが，Coq と違い各規則が独立したタクティックに対応している訳ではない．ほとんどのものが「move+修飾子」という形で書ける．

もう一回 modus_ponens を証明する．

```
Reset modus_ponens.                                  (* module_ponens 以降の定義を忘れる *)
Require Import ssreflect.                           (* ssreflect のタクティック言語を使う *)

Theorem modus_ponens : P -> (P -> Q) -> Q.
Proof.
  move=> p.                                                          (* Intro *)
1 subgoal
  p : P
  ============================
  (P -> Q) -> Q

  move/(_ p).                                                        (* Apply-L2 *)
1 subgoal
  p : P
  ============================
  Q -> Q

  done.                                                          (* 公理や矛盾 *)
No more subgoals.                                                  (* 証明完了 *)
Restart.                                                      (* 証明をやりなおす *)
  by move => p /(_ p).                                      (* 全てを一行にまとめる *)
Qed.
```

move の => と / は一緒にまとめられ，さらに by を他のタクティックの前に書くと，後に done を加えたときと同じ意味になる．

もう一つの恒真も試してみよう．

```
Theorem and_comm : P /\ Q -> Q /\ P.
Proof.
  move=> [] p q.                                          (* ∧-L, Intro, Intro *)
  p : P
  q : Q
  ============================
  Q /\ P

  split.                                                             (* ∧-R *)
2 subgoals

  P, Q : Prop
  p : P
  q : Q
  ============================
  Q

subgoal 2 is:
 P

  done. done.
```

```
No more subgoals.

Restart.
  by move=> [p q]; split.                (* 名前を中に書く，タクティックを「;」でつなぐ *)
Qed.
```

恒真色々　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．

```
Theorem or_comm : P \/ Q -> Q \/ P.
Proof.
  move=> [p|q].                    (* (∨-L; 証明木が分岐するとき，縦棒「|」で分ける *)
    by right.                                                    (* ∨-R2 *)
  by left.                                                       (* ∨-R1 *)
Qed.

Theorem DeMorgan : ~ (P \/ Q) -> ~ P /\ ~ Q.
Proof.
  rewrite /not.                                          (* 定義を展開する *)
    P, Q : Prop
    ============================
    (P ∨ Q -> False) -> (P -> False) ∧ (Q -> False)
  move=> npq.
  split=> [p|q].                              (* 全てのタクティックで「=>」が使える *)
    apply: npq.                                                  (* (Apply-R) *)
    by left.
  apply: npq.
  by right.
Qed.
End Koushin.                                                   (* Section を閉じる *)
```

論理和に関する De Morgan の法則が証明できた．しかし，双対的な法則 $(\neg(P \wedge Q) \supset \neg P \vee \neg Q)$ は直観主義論理ではなりたたない．二重否定の除去か排中律を仮定する必要がある．それらの同値性を証明した上でその定理を証明する．

```
Section Classic.
Definition Classic := forall P : Prop, ~ ~ P -> P.          (* ¬¬ 除去の定義 *)
Definition EM := forall P : Prop, P \/ ~ P.                  (* 排中律の定義 *)

Lemma Classic_is_EM : Classic <-> EM.                  (* ¬¬ 除去と排中律が同値 *)
Proof.
rewrite /Classic /EM.                                          (* 定義の展開 *)
split => [classic | em] P.                    (* A <-> B := A -> B /\ B -> A *)
  - apply: (classic) => nEM.              (* classic を仮定から消さずに焦点におく *)
    have p : P.                                                  (* (カット) *)
      apply: classic => np.
      apply: nEM. by right.
    apply: nEM. by left.
  - move: (em P) => [].                    (* P についての排中律で場合分けをする *)
    + done.
    + move => np /(_ np). done.
Qed.
```

Definition は項や命題を定義する．ここでは二重否定除去と排中律の言明を定義するのに使う．forall P : Prop はこの言明が任意の命題に対して使えることを表している．

証明の中の「-」(または「+」や「*」) は場合分けの構造を分かりやすくするために使う．一時的に他のゴールが見えなくなる．

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

```
Theorem DeMorgan' : Classic -> ~ (P /\ Q) -> ~ P \/ ~ Q.
Proof.
  move=> /Classic_is_EM em npq.                    (* Apply-L1, Intro, Intro *)
  move: (em P) => [p|np].                                (* 排中律で場合分け *)
  - move: (em Q) => [q|nq].
    + elim: npq.                                                    (* 矛盾 *)
      by split.
    + by right.
  - by left.
Qed.
End Classic.                                              (* Section を閉じる *)
Check DeMorgan'.                                    (* 命題の言明を確認する *)
DeMorgan'
     : forall P Q : Prop, Classic -> ~ (P /\ Q) -> ~ P \/ ~ Q
```

Section から出ると，使われた命題変数が各定理の言明に自動で追加される．

論理規則と tactic の対応

論理規則	作戦	論理規則	作戦
Intro	move => h	公理	done, by
Revert	move: h	矛盾	elim, suff: False
Apply-R	apply	$\wedge$-R	split
Apply-L1	move/e	$\wedge$-L	move => []
Apply-L2	move/(_ e)	$\vee$-R	left, right
カット	have: P, suff: P, move: (e)	$\vee$-L	move => [|]

h は仮定の名前，e は仮定を適用で組み合わせた証明式，P は論理式である．各タクティックの詳しい説明は次回 (述語論理) のときに述べる．

練習問題 2.1 以下の定理を Coq で証明せよ．

```
Section Coq1.
  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.
  Definition DM_rev := forall P Q, ~ (~P /\ ~Q) -> P \/ Q.
  Theorem DM_rev_is_EM : DM_rev <-> EM.
End Coq1.
```

3 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)

$$
\begin{array}{lcll}
t & ::= & x & \text{項変数} \\
 & | & c & \text{項定数} \\
 & | & f(t_1,\ldots,t_n) & \text{項関数} \\
\phi & ::= & p & \text{述語名} \\
 & | & X & \text{述語変数} \\
v & ::= & x \mid X & \text{変数} \\
\sigma & ::= & t \mid \phi & \text{代入}
\end{array}
\qquad
\begin{array}{lcll}
P,Q & ::= & \ldots & \text{命題} \\
 & | & \phi & \text{命題述語} \\
 & | & \phi(t_1,\ldots,t_n) & \text{述語} \\
 & | & \forall v.P & \text{全称} \\
 & | & \exists v.P & \text{存在} \\
 & | & t_1 = t_2 & \text{等価性}
\end{array}
$$

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**　$\dfrac{\Delta \vdash P \quad v \notin fv(\Delta)}{\Delta \vdash \forall v.P}$

$\forall$ **除去**　$\dfrac{\Delta \vdash \forall x.P}{\Delta \vdash [t/x]P} \quad \dfrac{\Delta \vdash \forall X.P}{\Delta \vdash [\phi/X]P}$

反射率　$\Delta \vdash t = t$

$\exists$ **導入**　$\dfrac{\Delta \vdash [t/x]P}{\Delta \vdash \exists x.P} \quad \dfrac{\Delta \vdash [\phi/X]P}{\Delta \vdash \exists X.P}$

$\exists$ **除去**　$\dfrac{\Delta \vdash \exists v.P \quad \Delta, P \vdash Q \quad v \notin fv(\Delta, Q)}{\Delta \vdash Q}$

代入　$\dfrac{\Delta \vdash [t_1/x]P \quad \Delta \vdash t_1 = t_2}{\Delta \vdash [t_2/x]P}$

ここで自由変数 $fv(P)$ と代入が利用される．

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

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

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

導出の例

$$
\dfrac{\dfrac{\forall x.\mathit{human}(x) \supset \mathit{mortal}(x) \vdash \forall x.\mathit{human}(x) \supset \mathit{mortal}(x)}{\forall x.\mathit{human}(x) \supset \mathit{mortal}(x) \vdash \mathit{human}(S) \supset \mathit{mortal}(S)} \qquad \mathit{human}(S) \vdash \mathit{human}(S)}{\forall x.\mathit{human}(x) \supset \mathit{mortal}(x), \mathit{human}(S) \vdash \mathit{mortal}(S)}
$$

3.1　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．

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

$$\vdash P : \mathit{Prop}$$

$$\vdash p, X : T_1 \to \ldots \to T_n \to \mathit{Prop}$$

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

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

$\Delta \vdash t:T$ 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 ($\vdash T : Prop$)，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.                                          (* 定義を表示する *)
ms = hm socrates hs                              (* ((hm socrates) hs) *)
     : mortal socrates
End Socrates.
```

3.2　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

```
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 Eq の transitivity と Group の inv_involutive を証明に変えよ.

3.3 全称と存在

∀ と ∃ の間に De Morgan の法則がなりたつ. 前回と同様に, ∃ 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 : (~ exists x, P x) -> forall x, ~ P x.
Proof.
  move=> N x Px. elim: N. by exists x.
Qed.

Theorem exists_or :
  (exists x, P x \/ Q x) -> (exists x, P x) \/ (exists x, Q x).
Proof.
  move=> [x [Px | Qx]]; [left|right]; by exists x.
Qed.

Hypothesis EM : forall P, P  ~P.

Lemma DeMorgan2' : ~ (forall x, P x) -> exists x, ~ P x.
Proof.
  move=> nap.
  case: (EM (exists x, ~ P x)) => //.                    (* 背理法 *)
```

```
  move=> nnpx.
  elim: nap => x.                                    (* (forall x, P x) を証明する *)
  case: (EM (P x)) => //.                                          (* 背理法 *)
  move=> npx.
  elim: nnpx.
  by exists x.
Qed.

End Laws.
```

練習問題 3.2 以下の定理を *Coq* で証明せよ.

```
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) \/ (exists x, Q x) -> exists x, P x \/ Q x.

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

  Variables InPub Drinker : A -> Prop.
  Theorem drinkers_paradox :
    (exists consumer, InPub consumer) ->
    exists man, InPub man /\ 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.
```

4 プログラミング言語としての Coq

Coq では，関数型言語のようにプログラムが書ける.
注意：Coq では各命令が”.” で終わる.

定義と関数

```
Definition one : nat := 1.                                        (* 定義 *)
one is defined

Definition one := 1.
Error: one already exists.                                  (* 再定義はできない *)

Definition one' := 1.                                    (* 型を書かなくてもいい *)
Print one'.                                                   (* 定義の確認 *)
one' = 1
     : nat                                                 (* nat は自然数の型 *)

Definition double x := x + x.                                     (* 関数の定義 *)
Print double.
```

```
double = fun x : nat => x + x                                   (* 関数も値 *)
     : nat -> nat                                               (* 関数の型 *)

Eval compute in double 2.                                       (* 式を計算する *)
     = 4
     : nat

Definition double' := fun x => x + x.                           (* 関数式で定義 *)
Print double'.
double' = fun x : nat => x + x
     : nat -> nat

Definition quad x := let y := double x in 2 * y.                (* 局所的な定義 *)
Eval compute in quad 2.
     = 8
     : nat

Definition quad' x := double (double x).                        (* 関数適用の入れ子 *)
Eval compute in quad' 2.
     = 8
     : nat

Definition triple x :=
  let double x := x + x in                                      (* 局所的な関数定義。上書きもできる *)
  double x + x.
Eval compute in triple 3.
     = 9
     : nat
```

計算の仕組み

Coq の計算がいくつかの簡約規則に基いています.

δ 簡約 証明の中で Definition を展開するために `rewrite /def` を使うが，`compute` はそれを勝手にやっている.

β 簡約 `(fun x => Body) Arg` のような関数適用は代入で評価される.

$$(\texttt{fun}\ x \Rightarrow B)\ A \longrightarrow [A/x]B$$

x が束縛変数なので，述語論理と同様に名前の衝突が回避される.

ζ 簡約 `let x := Def in Body` も同様に代入になる.

$$(\texttt{let}\ x := D\ \texttt{in}\ B \longrightarrow [D/x]B$$

ι 簡約 `(fix f x => Body) Arg` の場合，Arg を代入するだけでなく，`(fix f ...)` 自身を `f` に代入する.

$$(\texttt{fix}\ f\ x \Rightarrow B)\ A \longrightarrow [A/x, (\texttt{fix}\ f\ x \Rightarrow B)/f]B$$

データ型の定義

```
Inductive janken : Set :=                                       (* じゃんけんの手 *)
  | gu
  | choki
  | pa.

Definition weakness t :=                                        (* 弱点を返す *)
  match t with                                                  (* 簡単な場合分け *)
```

```
  | gu    => pa
  | choki => gu
  | pa    => choki
  end.
Eval compute in weakness pa.
     = choki
     : janken

Print bool.
Inductive bool : Set :=  true : bool | false : bool
Print janken.
Inductive janken : Set :=  gu : janken | choki : janken | pa : janken

Definition wins t1 t2 :=                          (* 「t1 は t2 に勝つ」という関係 *)
  match t1, t2 with                                      (* 二つの値で場合分け *)
  | gu, choki => true
  | choki, pa => true
  | pa,    gu => true
  | _, _      => false                                    (* 残りは全部勝たない *)
  end.
Check wins.
wins : janken -> janken -> bool                    (* 関係は bool への多引数関数 *)
Eval compute in wins gu pa.
     = false
     : bool
```

場合分けによる証明

```
Lemma weakness_wins t1 t2 :
  wins t1 t2 = true <-> weakness t2 = t1.
Proof.
  split.
  - by case: t1; case: t2.                                (* 全ての場合を考える *)
  - move=> <-; by case: t2.                              (* t2 の場合分けで十分 *)
Restart.
  case: t1; case: t2; by split.                    (* 最初から全ての場合でも OK *)
Qed.
```

再帰データ型と再帰関数

```
Module MyNat.                                          (* nat を新しく定義する *)
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined

Fixpoint plus (m n : nat) {struct m} : nat :=          (* 帰納法の対象を明示する *)
  match m with                                         (* 減らないとエラーになる *)
  | O => n
  | S m' => S (plus m n)
  end.
Error: Recursive definition of plus is ill-formed.
In environment ...
Recursive call to plus has principal argument equal to m instead of m'.

Fixpoint plus (m n : nat) {struct m} : nat :=          (* 同じ型の引数をまとめる *)
```

定義	`Definition f ... := ... .`
再帰的な定義	`Fixpoint f ... {struct x} := ... .`
データ型の定義	`Inductive t : Set := a | b : t -> t | c .`
局所的な定義	`let x := ... in ...`
局所関数	`fun x => ...`
局所再帰関数	`fix f ... {struct x} := ...`
if 文	`if ... then ... else ...`
場合分け	`match ... with` $\mathtt{pat}_1$ `=> ... | ... |` $\mathtt{pat}_n$ `=> ... end`

表 1: Coq の基本的な構文

```
  match m with
  | O => n
  | S m' => S (plus m' n)                                    (* 正しい定義 *)
  end.
plus is recursively defined (decreasing on 1st argument)

Print plus.
plus = fix plus (m n : nat) : nat := match m with
                                     | O => n
                                     | S m' => S (m' + n)
                                     end
     : nat -> nat -> nat

Check plus (S (S O)) (S O).                                  (* 式の型を調べる *)
plus (S (S O)) (S O)
     : nat

Eval compute in plus (S (S O)) (S O).                        (* 式を評価する *)
     = S (S (S O))
     : nat

Fixpoint mult (m n : nat) struct m : nat := O.

Eval compute in mult (S (S O)) (S O).
     = S (S O)                                               (* 期待している値 *)
     : nat
End MyNat.
```

練習問題 4.1 `mult` を正しく定義せよ.

5 帰納法による証明

Coq でデータ型を定義すると，自動的に帰納法の原理が生成される.

```
Check nat_ind.
nat_ind
     : ∀ P : nat -> Prop,
       P O ->
       (∀ n : nat, P n -> P (S n)) ->
       ∀ n : nat, P n
```

もっと分かりやすく書くと，`nat_ind` の型は

$$\forall P, P\ 0 \to (\forall n, P\ n \to P\ (S\ n)) \to (\forall n, P\ n)$$

である．即ち P は 0 でなりたち，任意の n について P が n でなりたてば，$n+1$ でももなりたつことが証明できれば，任意の n について P がなりたつ．

算数の様々な性質を帰納法で証明できる．elim は焦点の型によって帰納法の原理を選び，それを結論に適用する．

```
Lemma plusnS m n : m + S n = S (m + n).                          (* m, n は仮定 *)
Proof.
  elim: m => /=.                                                 (* nat_ind を使う *)
  - done.                                                          (* O の場合 *)
  - move => m IH.                                                  (* S の場合 *)
    by rewrite IH.                                        (* 帰納法の仮定で書き換える *)
Restart.
  elim: m => /= [|m ->] //.                                      (* 一行にまとめた *)
Qed.
Check plusnS.                                  (* ∀ m n : nat, m + S n = S (m + n) *)

Lemma plusSn m n : S m + n = S (m + n).
Proof. rewrite /=. done. Show Proof. Qed.              (* 簡約できるので帰納法は不要 *)

Lemma plusn0 n : n + 0 = n.
Admitted.                                                 (* 定理を認めて証明を終わらせる *)
Lemma plusC m n : m + n = n + m.
Admitted.
Lemma plusA m n p : m + (n + p) = (m + n) + p.
Admitted.

Lemma multnS m n : m * S n = m + m * n.
Proof.
  elim: m => /= [|m ->] //.
  by rewrite !plusA [n + m]plusC.
Qed.

Lemma multn0 n : n * 0 = 0.
Admitted.
Lemma multC m n : m * n = n * m.
Admitted.
Lemma multnDr m n p : (m + n) * p = m * p + n * p.
Admitted.
Lemma multA m n p : m * (n * p) = (m * n) * p.
Admitted.

Fixpoint sum n :=
  if n is S m then n + sum m else 0.
Print sum.                                  (* if .. is は match .. with に展開される *)

Lemma double_sum n : 2 * sum n = n * (n + 1).
Admitted.
Lemma square_eq a b : (a + b) * (a + b) = a * a + 2 * a * b + b * b.
Admitted.                                                  (* 帰納法なしで証明できる *)
```

練習問題 5.1 付録 A を参考にし，上の Admitted を全て証明せよ．

6 プログラムの型付け

6.1 型と型付け

型　　$\tau, \theta ::= \mathsf{nat} \mid \mathsf{Z} \mid \ldots \mid \theta \to \tau \mid \tau \times \theta$　　データ型, 関数型, 直積

型判定　$\Gamma \vdash M : \tau$　　$\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ という仮定のもとで，M が型 τ をもつ．

型付け規則　Coq の式は以下の型付け規則によって型付けされる．

変数　$\Gamma \vdash x : \tau$　$(x : \tau$はΓに含まれる$)$

抽象　$$\frac{\Gamma, x : \theta \vdash M : \tau}{\Gamma \vdash \mathtt{fun}\ x{:}\theta \Rightarrow M : \theta \to \tau}$$

適用　$$\frac{\Gamma \vdash M : \theta \to \tau \quad \Gamma \vdash N : \theta}{\Gamma \vdash M\ N : \tau}$$

定義　$$\frac{\Gamma \vdash M : \theta \quad \Gamma, x : \theta \vdash N : \tau}{\Gamma \vdash \mathtt{let}\ x\ \mathtt{:=}\ M\ \mathtt{in}\ N : \tau}$$

不動点　$$\frac{\Gamma, f : \theta \to \tau, x : \theta \vdash M : \tau}{\Gamma \vdash \mathtt{fix}\ f\ (x{:}\theta)\ \mathtt{:=}\ M : \theta \to \tau}$$

直積　$$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \theta}{\Gamma \vdash (M, N) : \tau \times \theta}$$

射影　$\Gamma \vdash \mathsf{fst} : \tau \times \theta \to \tau$　　$\Gamma \vdash \mathsf{snd} : \tau \times \theta \to \theta$

型付けの例

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \mathsf{nat} \vdash \mathsf{S} : \mathsf{nat} \to \mathsf{nat} \quad \Gamma, x : \mathsf{nat} \vdash x : \mathsf{nat}}{\Gamma, x : \mathsf{nat} \vdash \mathsf{S}\ x : \mathsf{nat}}\ \text{適用}}{\Gamma \vdash \mathtt{fun}\ x{:}\mathsf{nat} \Rightarrow \mathsf{S}\ x : \mathsf{nat} \to \mathsf{nat}}\ \text{抽象} \qquad \Gamma \vdash \mathsf{O} : \mathsf{nat}}{\Gamma \vdash (\mathtt{fun}\ x{:}\mathsf{nat} \Rightarrow \mathsf{S}\ x)\ \mathsf{O} : \mathsf{nat}}\ \text{適用}$$

6.2　命題と型の対応

カリー・ハワード同型により，命題論理と型理論 (型付 λ 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 Δ	型環境 Γ
$\supset$	$\to$
$\wedge$	$*$

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 $\langle \_ \rangle$: 命題 $\to$ 型 が存在し，任意の Δ と P について，導出 Π より $\Delta \vdash P$ が示せるならば，Π からプログラム M が作れ，$\langle \Delta \rangle \vdash M : \langle P \rangle$．また，任意の Γ, M, τ 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，上の**不動点**の規則は矛盾を生んでしまう．具体的には，$\theta = \mathit{True}$ と $\tau = \mathit{False}$ にすると，以下の導出が可能になる．

$$\frac{\Gamma, f : \mathit{True} \to \mathit{False}, x : \mathit{True} \vdash f\ x : \mathit{False}}{\Gamma \vdash \mathtt{fix}\ f\ (x{:}\theta)\ \mathtt{:=}\ f\ x : \mathit{True} \to \mathit{False}}$$

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**直感主義論理**に基いているからである．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．

7　帰納的な定義

7.1　Coq の帰納的データ型

前回は自然数の定義を見た．

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

実は，Coq の全てのデータは帰納的データ型として定義される.[1]

```
Inductive prod (A B : Set) : Set := pair : A -> B -> prod A B.

Inductive sum (A B : Set) : Set :=  inl : A -> sum A B | inr : B -> sum A B.
```

帰納的データ型の値を作るのは構成を適用するだけでいい．しかし，分解するのに OCaml と同様にパターンマッチングを使わなければならない．その型付け規則が複雑になる．以下のようなデータ型を考える．

$$
\begin{array}{ll}
\texttt{Inductive} & t(a_1 \ldots a_n : \texttt{Set}) : \texttt{Set} := \\
& \mid \quad c_1 : \tau_{11} \to \ldots \to \tau_{1k_1} \to t\ a_1 \ldots a_n \\
& \ldots \\
& \mid \quad c_m : \tau_{m1} \to \ldots \to \tau_{mk_m} \to t\ a_1 \ldots a_n.
\end{array}
$$

マッチング

$$
\frac{\Gamma \vdash M : t\ b_1 \ldots b_n \qquad \Gamma, x_{i1} : \tau_{i1}[b_1/a_1, \ldots, b_n/a_n], \ldots, x_{ik_i} : \tau_{ik_i}[\ldots] \vdash M_i : \tau[c_i\, x_{i1} \ldots x_{ik_i}/x] \quad (1 \le i \le m)}{\Gamma \vdash \mathsf{match}\ M\ \mathsf{as}\ x\ \mathsf{return}\ \tau\ \mathsf{with}\ c_1\, x_{11} \ldots x_{1k_1} \Rightarrow M_1 \mid \ldots \mid c_m\, x_{m1} \ldots x_{mk_m} \Rightarrow M_m\ \mathsf{end} : \tau[M/x]}
$$

as と return によって，返り値の型の中に入力を含めることができ，場合によって型が違うような関数が作れる．それを手でやるのは難しいが，作戦 case はこのパターンマッチングを構築してくれる．

帰納的データ型を定義すると，帰納法のための補題が自動的に定義されるが，定義は match を使う．定義が再帰的でないとき，パターンマッチングだけで済む．再帰的なデータ型について Fixpoint が使われる．

```
Definition prod_ind (A B:Set) (P:prod A B -> Prop) :=
  fun (f : forall a b, P (pair a b)) =>
  fun p => match p as x return P x with pair a b => f a b end.
Check prod_ind.
  : ∀ (A B : Set) (P : A * B -> Prop),
      (∀ (a : A) (b : B), P (a, b)) -> ∀ p : A * B, P p

Definition sum_ind (A B:Set) (P:sum A B -> Prop) :=
  fun (fl : forall a, P (inl _ a)) (fr : forall b, P (inr _ b)) =>
  fun p => match p as x return P x
           with inl a => fl a | inr b => fr b end.
Check sum_ind.
  :  ∀ (A B : Set) (P : A + B -> Prop),
       (∀ a : A, P (inl B a)) -> (∀ b : B, P (inr A b)) ->
       ∀ p : A + B, P p

Fixpoint nat_ind (P:nat -> Prop) (f0:P O) (fn:forall n, P n -> P (S n))
  (n : nat) {struct n} :=
  match n as x return P x
  with O => f0 | S m => fn m (nat_ind P f0 fn m) end.
Check nat_ind.
  : ∀ P : nat -> Prop, P O -> (∀ n : nat, P n -> P (S n)) ->
      ∀ n : nat, P n
```

case と elim はよく似ているが，前者が単なる場合分けを行うのに対して，後者が生成された補題を利用しているので，効果が違ったりする．

```
Lemma plusn0 n : n + 0 = n.
```

[1] 実際の定義を見ると，Set ではなく Type になっている．Type は Set より一般的なので，Set として使うことができる．さらに，prod A B は A*B として表示され，pair a b は (a,b) として表示される．Coq の Notation という機能によって，機能的データ型の表示方法を変えることができる．

```
Proof.
  case: n.
  - done.
∀ n : nat, S n + 0 = S n
Restart.
  move: n.
  apply: nat_ind.                                          (* elim の意味 *)
  - done.
∀ n : nat, n + 0 = n -> S n + 0 = S n
  - move=> n /= -> //.
Qed.
```

7.2 帰納的述語

Coq では帰納的な定義は Set だけでなく Prop でもできる．この場合，パラメータは場合によって変わることが多い．

$$\begin{array}{rl} \texttt{Inductive} & t : \tau_1 \to \ldots \to \tau_n \to \texttt{Prop} := \\ | & c_1 : \forall (x_1 : \tau_{11}) \ldots (x_{k_1} : \tau_{1k_1}), t\ \theta_{11} \ldots \theta_{1n} \\ & \ldots \\ | & c_m : \forall (x_1 : \tau_{m1}) \ldots (x_{k_1} : \tau_{mk_m}), t\ \theta_{m1} \ldots \theta_{mn} \end{array}$$

マッチング

$$\frac{\Gamma \vdash M : t\ b_1 \ldots b_n \qquad \Gamma, x_{i1} : \tau_{i1}, \ldots, x_{ik_i} : \tau_{ik_i} \vdash M_i : \tau[\theta_{i1} \ldots \theta_{in}/x_1 \ldots x_n] \quad (1 \le i \le m)}{\begin{array}{l}\Gamma \vdash \mathsf{match}\ M\ \mathsf{in}\ t\ x_1 \ldots x_n\ \mathsf{return}\ \tau\ \mathsf{with}\ c_1\, x_{11} \ldots x_{1k_1} \Rightarrow M_1 \mid \ldots \mid c_m\, x_{m1} \ldots x_{mk_m} \Rightarrow M_m\ \mathsf{end} \\ \qquad : \tau[b_1, \ldots, b_n/x_1 \ldots x_n]\end{array}}$$

```
(* 偶数の定義 *)
Inductive even : nat -> Prop :=
  | even_O : even O
  | even_SS : forall n, even n -> even (S (S n)).

(* 帰納的述語を証明する定理 *)
Theorem even_double n : even (n + n).
Proof.
  elim: n => /= [|n IH].
  - apply: even_O.
  - rewrite -plus_n_Sm.
    by apply: even_SS.
Qed.

(* 帰納的述語に対する帰納法もできる *)
Theorem even_plus m n : even m -> even n -> even (m + n).
Proof.
  elim: m => //=.
Restart.
  move=> Hm Hn.
  elim: Hm => //= m m IH.
  apply: even_SS.
Qed.

(* 矛盾を導き出す *)
Theorem one_not_even : ~ even 1.
Proof.
  case.
Restart.
  move H: 1 => one He.  (* move H: exp => pat は H: exp = pat を作る *)
  case: He H => //.
```

```
Restart.
  move=> He.
  inversion He.
  Show Proof.  (* 証明が複雑で、SSReflect では様々な理由で避ける *)
Qed.

(* 等式を導き出す *)
Theorem eq_pred m n : S m = S n -> m = n.
Proof.
  case.  (* 等式を分解する *)
  done.
Qed.
```

実は Coq の論理結合子のほとんどが帰納的述語として定義されている.

```
Inductive and (A B : Prop) : Prop :=  conj : A -> B -> A /\ B.

Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B.

Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : forall x : A, P x -> exists x, P x.

Inductive False : Prop := .
```

and, or や ex について case が使えた理由がこの定義方法である.

しかも, False は最初からあるものではなく, 構成子のない述語として定義されている. 生成される帰納法の補題をみると面白い.

```
Print False_ind.
fun (P : Prop) (f : False) => match f return P with end
     : ∀ P : Prop, False -> P
```

ちょうど, 矛盾の規則に対応している. 作戦　elim でそれが使える.

```
Theorem contradict (P Q : Prop) : P -> ~P -> Q.
Proof. move=> p. elim. exact: p. Qed.
```

練習問題 7.1 以下の定理を証明しなさい.

```
Module Odd.
Inductive odd : nat -> Prop :=
  | odd_1 : odd 1
  | odd_SS : forall n, odd n -> odd (S (S n)).

Theorem even_odd n : even n -> odd (S n). Abort.
Theorem odd_even n : odd n -> even (S n). Abort.
Theorem even_not_odd n : even n -> ~odd n. Abort.
End Odd.
```

8 Mathcomp と数学の証明

8.1 MathComp のライブラリ

SSReflect のコマンドを既に見たが, MathComp の本当の強さはそのライブラリにある. その大きな特徴は書き換えを証明の基本手法とすること.

ライブラリは ssreflect, fingroup, algebra 等, いくつかのの部分からできている. 前者は一般的なデータ構造で, 後者は代数系の証明に使う.

Search　SSReflect の Search コマンドが強力で，ライブラリを探すのに便利.

```
Search "add".                                    (* 名前に add を含む定理を検索する *)
Search (_ + S _).                                (* 結論がパターンを含む定理を検索する *)
Search _ (_ + S _).                              (* 前提または結論がパターンを含む定理を検索する *)
Search (_ + _) (_ * _) "mul".                    (* 左を全てみたすものを検索する *)
```

8.2　基本データ

まず，ssreflect を読み込む.

```
From mathcomp Require Import all_ssreflect.
```

いくつかのモジュールが読み込まれます．ssrbool は論理式と述語の扱い．ssrnat は自然数．ssrfun は関数 (写像) の様々な性質．seq はリスト．eqtype，choice，fintype はそれぞれ等価性，選択，有限性が使えるデータ構造のための枠組みを提供している．例えば，自然数の等価性は判定できるので，排中律を仮定しなくても場合分けができる.

中身について，ファイルを参照するしかないが，まず ssrnat の例をみよう．ちなみに，ソースファイルはは Coq Platform の場合，Coq のリソースフォルダ (Windows で C:\Coq,
MacOS で/Applications/Coq_Platform_2021.02.1.app/Contents/Resources)
の下の lib/coq/user-contrib/mathcomp/ssreflect と lib/coq/theories/ssr の下にある.

```
Module Test_ssrnat.
Fixpoint sum n :=
  if n is m.+1 then n + sum m else 0.

Theorem double_sum n : 2 * sum n = n * n.+1.
Proof.
  elim: n => [|n IHn] //=.
  rewrite -[n.+2]addn2 !mulnDr.
  rewrite addnC !(mulnC n.+1).
  by rewrite IHn.
Qed.
End Test_ssrnat.
```

8.3　自己反映

論理式も書き換えで処理したい．そのために，ssrbool では論理演算子を型 bool の上の演算子として定義している．例えば，/\ は&&, \/ は||になる．二つの定義の間に行き来するために，reflect という自己反映を表した宣言を使う．それが SSReflect の名前の由来である.

```
Print reflect.
Inductive reflect (P : Prop) : bool -> Set :=
    ReflectT : P -> reflect P true | ReflectF : ~ P -> reflect P false
Check orP.
orP : ∀ b1 b2 : bool, reflect (b1  b2) (b1 || b2)
```

表現の切り替えはビュー機構によって行われる．前に見た適用パターンを使う．move, case, apply などの直後に/view を付けると，対処が可能な方向に変換される．=>の右でも使える．なお，ビューとしては上の reflect 型 だででなく，同値関係 (P <-> Q) や普通の定理 (P -> Q) も使える.

```
Module Test_ssrbool.
Variables a b c : bool.

Print andb.
```

```
Lemma andb_intro : a -> b -> a && b.
Proof.
  move=> a b.
  rewrite a.
  move=> /=.
  done.
Restart.
  by move ->.
Qed.

Lemma andbC : a && b -> b && a.
Proof.
  case: a => /=.
    by rewrite andbT.
  done.
Restart.
  by case: a => //= ->.
Restart.
  by case: a; case: b.
Qed.

Lemma orbC : a || b -> b || a.
Proof.
  case: a => /=.
    by rewrite orbT.
  by rewrite orbF.
Restart.
  move/orP => H.
  apply/orP.
  move: H => [Ha|Hb].
    by right.
  by left.
Restart.
  by case: a; case: b.
Qed.

Lemma test_if x : if x == 3 then x*x == 9 else x !=3.
Proof.
  case Hx: (x == 3).
    by rewrite (eqP Hx).
  done.
Restart.
  case: ifP.
    by move/eqP ->.
  move/negbT. done.
Qed.
End Test_ssrbool.
```

自己反映があると自然数の証明もスムーズになる.

```
Theorem avg_prod2 m n p : m+n = p+p -> (p - n) * (p - m) = 0.
Proof.
  move=> Hmn.
  have Hp0 q: p <= q -> p-q = 0.
    rewrite -subn_eq0. by move/eqP.
  suff /orP[Hpm|Hpn]: (p <= m) || (p <= n).
    - by rewrite (Hp0 m) // muln0.
    - by rewrite (Hp0 n).
  case: (leqP p m) => Hpm //=.
  case: (leqP p n) => Hpn //=.
```

```
    suff: m + n < p + p.
      by rewrite Hmn ltnn.
    by rewrite -addnS leq_add // ltnW.
  Qed.
```

練習問題 8.1 以下の等式を証明しなさい．タクティクは rewrite のみでできる．
ssrnat_doc.v の補題でほぼ足りるが, leq_mul も便利.

```
Module Equalities.
  Theorem square_sum a b : (a + b)^2 = a^2 + 2 * a * b + b^2. Abort.
  Theorem diff_square m n : m >= n -> m^2 - n^2 = (m+n) * (m-n). Abort.
  Theorem square_diff m n : m >= n -> (m-n)^2 = m^2 + n^2 - 2 * m * n. Abort.
End Equalities.
```

8.4 最大公約数の計算

ユークリッドが発明した互除法による最大公約数の計算は多分世界最古のアルゴリズムの一つである．その正しさを証明する．

```
let rec gcd m n =
  if m = 0 then n else gcd (n mod m) m
```

最大公約数の厳密な定義は

$$q \text{ は } m \text{ と } n \text{ の最大公約数である} \Leftrightarrow \begin{cases} q \mid m \ \wedge \ q \mid n \\ \forall q', (q' \mid m \ \wedge \ q' \mid n) \Rightarrow q' \mid q \end{cases}$$

二つ目に関して，本来は $q' \leq q$ のはずだが，$q' \mid q$ の方が証明しやすい．証明は m に関する簡単な帰納法である．

上の定義を Coq に与えると問題が生じる．

```
Fail Fixpoint gcd (m n : nat) {struct m} : nat :=
  if m is 0 then n else gcd (n %% m) m.
Error:
Recursive definition of gcd is ill-formed.
Recursive call to gcd has principal argument equal to
"n %% m" instead of "n0".
```

どうも，Coq が n %% m が m より小さいことを理解していないようだ．解決法は 2 つある．

ダミーの引数 常に m より大きいダミーの引数を追加して，その引数に対する帰納法を使う．

```
Fixpoint gcd (h m n : nat) {struct h} : nat :=
  if h is h.+1 then
    if m is 0 then n else gcd h (n %% m) m
  else 0.
```

h に関する場合分けが常に成功する（h が 0 になることはない）ことを証明しなければならないが，難しくはない．しかし，このやり方を使うと，Extraction の後でも h がコードの中に残り，本来のアルゴリズムと少し違ってしまう．

整礎帰納法 整礎な順序とは，無限な減少列を持たない順序のことを言う．自然数の上では $<$ は整礎である．特定の引数が全ての再帰呼び出しで整礎な順序において減少しているならば，関数の計算が無限に続くことはないので，Coq が定義を認める．(実際には減少の証明の構造に関する構造的帰納法が使われている)

Fixpoint の代わりに Function を使い，struct（構造）を wf（整礎）に変える．この方法では，定義と同時に引数が小さくなることを証明しなければならない．

```
Require Import Wf_nat Recdef.
Check lt_wf.
     : well_founded lt
Check lt_wf_ind.
     : ∀n (P : nat → Prop), (∀n', (∀m, m < n' → P m) → P n') → P n

Function gcd (m n : nat) {wf lt m} : nat :=
  if m is 0 then n else gcd (modn n m) m.
Proof.
- move=> m n m0 _. apply/ltP.
  by rewrite ltn_mod.
- exact: lt_wf.
Qed.
gcd_ind is defined
...
gcd is defined
gcd_equation is defined
Check gcd_equation.
Check gcd_ind.
Print gcd_terminate.

Require Import Extraction.
Extraction gcd.                                          (* wf が消える *)
let rec gcd m n =
  match m with
  | O -> n
  | S n0 -> gcd (modn n (S n0)) (S n0)
```

では，これから正しさを証明する．

```
Search (_ %| _) "dvdn".                          (* 割り切ることに関する補題を表示 *)
Check divn_eq.
     : ∀ m d : nat, m = m %/ d * d + m %% d

Theorem gcd_divides m n : (gcd m n %| m) && (gcd m n %| n).
Proof.
  functional induction (gcd m n).
    by rewrite dvdn0 dvdnn.
Admitted.

Check addKn.
     : ∀ x y : nat, x + y - x = y

Theorem gcd_max g m n : g %| m -> g %| n -> g %| gcd m n.
Admitted.
```

8.5 $\sqrt{2}$ が無理数

まずは自然数で以下の定理を証明する．

定理 2 任意の自然数 n と p について，

$$n \cdot n = 2(p \cdot p) \text{ ならば } p = 0$$

証明は n の関する整礎帰納法を使う．

- $n = 0$ のとき，$p = 0$
- $n \neq 0$ のとき，

- n と p が偶数でなければならないので，$n = 2n', p = 2p'$ とおける
- 再び，$n' \cdot n' = 2(p' \cdot p')$ が得られ，$n' < n$
- 帰納法の仮定より $p' = 0$
- すなわち，$p = 0$

その定理を使って，$\sqrt{2}$ が無理数であることを証明する．もしも $\sqrt{2}$ が有理数なら，ある n と p が存在し，$\sqrt{2} = n/p$，すなわち $n^2 = 2p^2$．しかし上の定理から $p = 0$ となるので矛盾．

実際に証明する．

```
odd_mul    : ∀ m n : nat, odd (m * n) = odd m && odd n
odd_double : ∀ n : nat, odd n.*2 = false
odd_double_half : ∀ n : nat, odd n + (n./2).*2 = n
andbb      : ∀ x : bool, x && x = x
negbTE     : ∀ b : bool, ~~ b -> b = false
double_inj : ∀ x x2 : nat, x.*2 = x2.*2 -> x = x2
divn2      : ∀ m : nat, m %/ 2 = m./2
ltn_Pdiv   : ∀ m d : nat, 1 < d -> 0 < m -> m %/ d < m
muln2      : ∀ m : nat, m * 2 = m.*2
esym       : ∀ (A : Type) (x y : A), x = y -> y = x

Lemma odd_square n : odd n = odd (n*n). Admitted.
Lemma even_double_half n : ~~odd n -> n./2.*2 = n. Admitted.


(* 本定理 *)
Theorem main_thm (n p : nat) : n * n = (p * p).*2 -> p = 0.
Proof.
  elim/lt_wf_ind: n p => n.                                    (* 清楚帰納法 *)
  case: (posnP n) => [-> _ [] // | Hn IH p Hnp].
Admitted.

(* 無理数 *)
Require Import Reals Field.                    (* 実数とそのための field タクティク *)

Definition irrational (x : R) : Prop :=
   forall (p q : nat), q <> 0 -> x <> (INR p / INR q)%R.

Theorem irrational_sqrt_2: irrational (sqrt (INR 2)).
Proof.
  move=> p q Hq Hrt.
  apply /Hq  /(main_thm p) /INR_eq.
  rewrite -mul2n !mult_INR -(sqrt_def (INR 2)) ?Hrt; last by auto with real.
  have Hqr : INR q <> 0%R by auto with real.
  by field.
Qed.
```

練習問題 8.2 `Admitted` を `Qed` に変え，証明を完成せよ．

9 プログラムの証明

9.1 依存和

存在 (∃) は帰納型の特殊な例である．

```
Print ex.
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : ∀ x : A, P x -> ex P.
```

ex (fun x:A => P(x)) を exists x:A, P(x) と書いてもいい.

この定義を見ると，ex P $= \exists x, P(x)$ は x と $P(x)$ の対でしかない．対の第 2 要素に第 1 要素が現れているので，この積を「依存和」という.（元々依存のある関数型を定義域を添字とした依存積と見なすなら，こちらは A を添字とする直和集合になる）

既に見ているように，証明の中で依存和を構築する時に，exists という作戦を使う.

```
Lemma exists_pred x : x > 0 -> exists y, x = S y.
Proof.
  case x => // n _.                                  (* case: x と同じだが項が読みやすい *)
  by exists n.
Qed.
Print exists_pred.
exists_pred =
fun x : nat =>
match x as n return (0 < n -> exists y : nat, n = y.+1) with
| 0 =>
    fun H : 0 < 0 =>
    let H0 : False :=
      eq_ind (0 < 0) (fun e : bool => if e then False else True) I true H in
    False_ind (exists y : nat, 0 = y.+1) H0                    (* 0 はありえない *)
| n.+1 =>
    fun _ : 0 < n.+1 => ex_intro (fun y : nat => n.+1 = y.+1) n (erefl n.+1)
end                                                  (* n.+1 のとき n を返す *)
     : ∀ x : nat, 0 < x -> exists y : nat, x = y.+1
Require Extraction.
Extraction exists_pred.                                    (* 何も抽出されない *)
```

上記の ex は Prop に住むものなので，論理式の中でしか使えない．しかし，プログラムの中で依存和を使いたい時もある．この時には sig を使う.

```
Print sig.
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : ∀ x : A, P x -> sig P.
```

sig (fun x:T => Px) は {x:T | Px} とも書く．ex と同様に，具体的な値は exists で指定する.

こういう条件付きな値を扱う安全な関数が書ける.

```
Definition safe_pred x : x > 0 -> {y | x = S y}.
  case x => // n _.                                    (* exists_pred と同じ *)
  by exists n.                                  (* こちらも exists を使う *)
Defined.                                (* 定義を透明にし，計算に使えるようにする *)
```

証明された関数を OCaml の関数として輸出できる．その場合，Prop の部分が消される.

```
Require Extraction.
Extraction safe_pred.
(** val safe_pred : nat -> nat **)
let safe_pred = function
  | O -> assert false (* absurd case *)
  | S x' -> x'
```

9.2 Hint と auto

証明が冗長になることが多い．auto は簡単な規則で証明を補完しようとする．具体的には，auto は 仮定や Hint Resolve lem1 lem2 ... で登録した定理を apply で適用しようとする．これらを組み合わせて，深さ 5 の項まで作れる (auto n で深さ n にできる)．info_auto で使われたヒントを表示させる事もできる.

Hint Constructors で帰納型を登録すると，各構成子が定理として登録される．また，auto using lem1，lem2，... で一回だけヒントを追加することもできる．

auto で定理が適用されるために，全ての変数が定理の結論に現れる必要がある．eauto を使うと simple apply が eapply に変わるので，決まらない変数が変数のまま残せる．その代わり，可能な導出木が増えるので，探索が中々終わらない場合もある．

9.3 整列の証明

```
Section Sort.
  Variables (A:Set) (le:A->A->bool).                    (* データ型 A とのその順序 le *)

  (* 既に整列されたリスト l の中に a を挿入する *)
  Fixpoint insert a (l: list A) :=
    match l with
    | nil => (a :: nil)
    | b :: l' => if le a b then a :: l else b :: insert a l'
    end.

  (* 繰り返しの挿入でリスト l を整列する *)
  Fixpoint isort (l : list A) : list A :=
    match l with
    | nil => nil
    | a :: l' => insert a (isort l')
    end.

  (* le は推移律と完全性をみたす *)
  Hypothesis le_trans: forall x y z, le x y -> le y z -> le x z.
  Hypothesis le_total: forall x y, ~~ le x y -> le y x.

  (* le_list x l : x はあるリスト l の全ての要素以下である *)
  Inductive le_list x : list A -> Prop :=
    | le_nil : le_list x nil
    | le_cons : forall y l,
        le x y -> le_list x l -> le_list x (y::l).

  (* sorted l : リスト l は整列されている *)
  Inductive sorted : list A -> Prop :=
    | sorted_nil : sorted nil
    | sorted_cons : forall a l,
        le_list a l -> sorted l -> sorted (a::l).

  Hint Constructors le_list sorted.                     (* auto の候補にする *)

  Lemma le_list_insert a b l :
    le a b -> le_list a l -> le_list a (insert b l).
  Proof.
    move=> leab; elim => {l} [|c l] /=. info_auto.
    case: ifPn. info_auto. info_auto.
  Qed.

  Lemma le_list_trans a b l :
    le a b -> le_list b l -> le_list a l.
  Proof.
    move=> leab; elim. info_auto.
    info_eauto using le_trans.                          (* 推移律は eauto が必要 *)
  Qed.

  Hint Resolve le_list_insert le_list_trans.            (* 補題も候補に加える *)
```

```
  Theorem insert_ok a l : sorted l -> sorted (insert a l). Admitted.
  Theorem isort_ok l : sorted (isort l). Admitted.

  (* Permutation l1 l2 : リスト l2 は l1 の置換である *)
  Inductive Permutation : list A -> list A -> Prop :=
    | perm_nil: Permutation nil nil
    | perm_skip: forall x l l',
        Permutation l l' -> Permutation (x::l) (x::l')
    | perm_swap: forall x y l, Permutation (y::x::l) (x::y::l)
    | perm_trans: forall l l' l'',
        Permutation l l' ->
        Permutation l' l'' -> Permutation l l''.

  Hint Constructors Permutation.

  Theorem Permutation_refl l : Permutation l l. Admitted.
  Theorem insert_perm l a : Permutation (a :: l) (insert a l). Admitted.
  Theorem isort_perm l : Permutation l (isort l). Admitted.

  (* 証明付き整列関数 *)
  Definition safe_isort l : {l'|sorted l' /\ Permutation l l'}.
    exists (isort l).
    auto using isort_ok, isort_perm.
  Defined.
  Print safe_isort.
End Sort.

Check safe_isort.                          (* le と必要な補題を与えなければならない *)
Extraction leq.                              (* mathcomp の eqType の抽出が汚ない *)

Definition leq' m n := if m - n is 0 then true else false.
Extraction leq'.                                          (* こちらはすっきりする *)

Lemma leq'E m n : leq' m n = (m <= n).
Proof. rewrite /leq' /leq. by case: (m-n). Qed.

Lemma leq'_trans m n p : leq' m n -> leq' n p -> leq' m p.
Proof. rewrite !leq'E; apply leq_trans. Qed.

Lemma leq'_total m n : ~~ leq' m n -> leq' n m. Admitted.

Definition isort_leq := safe_isort nat leq' leq'_trans leq'_total.

Eval compute in proj1_sig (isort_leq (3 :: 1 :: 2 :: 0 :: nil)).
     = [:: 0; 1; 2; 3] : seq nat

Extraction "isort.ml" isort_leq.
```

練習問題 9.1　*1.* Admitted を Proof に変え，証明を完成させよ．

2. le_list を以下のように一般化できる．

```
Inductive All (P : A -> Prop) : list A -> Prop :=
  | All_nil : All P nil
  | All_cons : forall y l, P y -> All P l -> All P (y::l).
```

このとき，All (le a) l が le_list a l と同じ意味になる．

こちらを使うように証明を修正せよ．

10 Mathcompで線形代数

Mathcomp の algebra フォルダが代数学関係のライブラリを与えている。

zmodp.v	$\mathbf{Z}/p\mathbf{Z}$
ssralg.v	環など
poly.v	多項式
ssrnum.v	体など
matrix.v	行列
vector.v	ベクトル空間

10.1 ベクトル空間

以下の問題を解きます。

1. E を K 上のベクトル空間とする。以下が同値であることを証明せよ。

$$E = \operatorname{Im} f \oplus \operatorname{Ker} f \quad \Leftrightarrow \quad \operatorname{Im} f = \operatorname{Im} (f \circ f)$$

2. E を K 上のベクトル空間とする。p と q を E 上の射影写像とする。

   (a) $p \circ q = q \circ p = 0$ が $p + q$ が射影写像である必要十分条件であることを証明せよ

   (b) $p + q$ が射影写像なら、以下が成り立つことを証明せよ

$$\operatorname{Im} (p + q) = \operatorname{Im} p \oplus \operatorname{Im} q$$

$$\operatorname{Ker} (p + q) = \operatorname{Ker} p \cap \operatorname{Ker} q$$

定義と方法

```
From mathcomp Require Import all_ssreflect all_algebra. (* 代数ライブラリ *)

Local Open Scope ring_scope. (* 環構造を使う *)
Import GRing.Theory.

Section Problem1.
Variable K : fieldType.                                          (* 体 *)
Variable E : vectType K.                                (* 有限次元ベクトル空間 *)
Variable f : 'End(E).                                        (* 線形変換 *)

Theorem equiv1 :
  (limg f + lker f)%VS = fullv <-> limg f = limg (f ø f).
Proof.
split.
- move/(f_equal (lfun_img f)).
  rewrite limg_comp limg_add.
  admit.
- rewrite limg_comp => Hf'.
  move: (limg_ker_dim f (limg f)).
  rewrite -[RHS]addOn -Hf' => /eqP.
  rewrite eqn_add2r dimv_eq0 => /eqP /dimv_disjoint_sum.
Admitted.
End Problem1.

Section Problem2.
```

```
Variable K : numFieldType.                                    (* ノルム付き体 *)
Variable E : vectType K.
Variable p q : 'End(E).

Definition projection (f : 'End(E)) := forall x, f (f x) = f x.

Lemma proj_idE f : projection f <-> {in limg f, f =1 id}.
Proof.
split => Hf x.
- by move/limg_lfunVK => <-.
- by rewrite Hf // memv_img ?memvf.
Qed.

Hypothesis proj_p : projection p.
Hypothesis proj_q : projection q.

Section a.
Lemma f_g_0 f g x :
  projection f -> projection g -> projection (f+g) -> f (g x) = 0.
Proof.
move=> Pf Pg /(_ (g x)).
rewrite !add_lfunE !linearD /=.
rewrite !Pf !Pg => /eqP.
rewrite -subr_eq !addrA addrK.
rewrite addrAC eq_sym -subr_eq eq_sym subrr => /eqP Hfg.
move: (f_equal g Hfg).
rewrite !linearD /= Pg linear0 => /eqP.
Admitted.

Theorem equiv2 :
  projection (p + q) <-> (forall x, p (q x) = 0 /\ q (p x) = 0).
Proof.
split=> H x.
Admitted.
End a.

Section b.
Hypothesis proj_pq : projection (p + q).

Lemma b1a x : x \in limg p -> x \in limg q -> x = 0.
Admitted.

Lemma b1b : directv (limg p + limg q).
Proof.
apply/directv_addP/eqP.
rewrite -subv0.
apply/subvP => u /memv_capP [Hp Hq].
rewrite memv0.
Admitted.

Lemma limg_sub_lker f g :
  projection f -> projection g -> projection (f+g) -> (limg f <= lker g)%VS.
Admitted.

Lemma b1c : (limg p <= lker q)%VS. Admitted.

Lemma b1c' : (limg q <= lker p)%VS. Admitted.

Lemma limg_addv (f g : 'End(E)) : (limg (f + g)%R <= limg f + limg g)%VS.
Proof.
```

```
apply/subvP => x /memv_imgP [u _ ->].
Admitted.

Theorem b1 : limg (p+q) = (limg p + limg q)%VS.
Proof.
apply/eqP; rewrite eqEsubv limg_addv /=.
apply/subvP => x /memv_addP [u Hu] [v Hv ->].
have -> : u + v = (p + q) (u + v).
  rewrite lfun_simp !linearD /=.
  rewrite (proj1 (proj_idE p)) // (proj1 (proj_idE q) _ v) //.
Admitted.

Theorem b2 : lker (p+q) = (lker p :&: lker q)%VS.
Proof.
apply/vspaceP => x.
rewrite memv_cap !memv_ker.
rewrite add_lfunE.
case Hpx: (p x == 0).
Admitted.
End b.
End Problem2.
```

Mathcomp の定理

```
(* ベクトル空間について *)
Lemma lkerE f U : (U <= lker f)%VS = (f @: U == 0)%VS.
Lemma subvv U : (U <= U)%VS.
Lemma subv0 U : (U <= 0)%VS = (U == 0%VS).
Lemma addv0 : right_id 0%VS addV.
Lemma capfv : left_id fullv capV.
Lemma subvf U : (U <= fullv)%VS.
Lemma memvf v : v \in fullv.
Lemma memvN U v : (- v \in U) = (v \in U).
Lemma memv_add u v U V : u \in U -> v \in V -> u + v \in (U + V)%VS.
Lemma memv_cap w U V : (w \in U :&: V)%VS = (w \in U) && (w \in V).
Lemma memv_img f v U : v \in U -> f v \in (f @: U)%VS.
Lemma memv_ker f v : (v \in lker f) = (f v == 0).
Lemma limg_ker_dim f U : (\dim (U :&: lker f) + \dim (f @: U) = \dim U)%N.
Lemma dimv_disjoint_sum U V :
  (U :&: V = 0)%VS -> \dim (U + V) = (\dim U + \dim V)%N.
Lemma dimv_eq0 U :  (\dim U == 0%N) = (U == 0%VS).
Lemma eqEdim U V : (U == V) = (U <= V)%VS && (\dim V <= \dim U).
Lemma eqEsubv U V : (U == V) = (U <= V <= U)%VS.
Lemma vspaceP U V : U =i V <-> U = V.

(* 環と体について *)
Lemma addr0 : right_id 0 +%R.
Lemma addrA : associative +%R.
Lemma addrC : commutative +%R.
Lemma subr_eq x y z : (x - z == y) = (x == y + z).
Lemma mulr2n x : x *+ 2 = x + x.
Lemma scaler_nat n v : n%:R *: v = v *+ n.
Lemma scaler_eq0 a v : (a *: v == 0) = (a == 0) || (v == 0).
Lemma linear0 (f : {linear U -> V | s}) : f 0 = 0.
Lemma linearD (f : {linear U -> V | s}) : {morph f : x y / x + y}.
Lemma Num.Theory.pnatr_eq0 n : (n%:R == 0 :> R) = (n == 0)%N.
```

A　タクティック・タクティカル・修飾子

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．

基本タクティック

move, apply, done, case, split, left, right, elim, have, suff, rewrite, set でほぼの証明ができるが, 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)

```
仮定 Δ
=================
焦点 P -> 結論 Q
```

各タクティックがゴールの各部 $(\Delta; P \vdash Q)$ 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，修飾子によって仮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: $\Delta; P_1 \to \ldots \to P_n \vdash Q$
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` $H : P$
新しい仮定 $H : P$ 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 $\Delta, H : P; \vdash Q$
仮定名 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．新しい仮定の元で現在のゴールを証明した後にその仮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` $x := t$
仮定に定義 $x := t : T$ を加える．x は仮定名, 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 は仮定名，t は項．Δ の元で t の型が T なら，仮定 $H : T$ 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: `:` $H_1 \ldots H_n$
後の状態: $\Delta'; P_1 \vdash P_2 \to \ldots \to P_n \to Q$
$H_i : P_i$ が Δ に含まれる仮定なら，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，不要になった仮定 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仮定に変える．

- `[]` 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 `|` `...|` 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 `|` `...|` 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，その後に焦点が消される．
- `<-` 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` $H_1 \ldots H_n$　ほぼ `:` $H_1 \ldots H_n$ `=>` $H_1 \ldots H_n$ と同じ意味だが，操作が元の結論に適用されない．

tac$_1$`;` *tac*$_2$　*tac*$_1$ を実行した後に，生成された全てのゴールに対して *tac*$_2$ を実行する．

tac`; [`*tac*$_1$ `|` `...|` *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 $tac_1, \ldots, tac_n$ 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(穴のある項) にマッチする最初の出現を書き換える.

/def 定義 *def* を展開する.

-/def 定義 *def* を畳み込む.

(*lem*$_1$,...,*lem*$_n$)
: *lem*$_1$... *lem*$_n$ 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.

(_ : t_1 = t_2)
: t_1 を t_2 に置き換える． t_1 = t_2 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.