「計算と論理」 Software Foundations その5

五十嵐 淳
cal22@fos.kuis.kyoto-u.ac.jp
http://www.fos.kuis.kyoto-u.ac.jp/~igarashi/class/cal/

京都大学

November 29, 2022

Tactics.v: 目次

Coq のタクティックについてさらに学ぼう

- `apply` タクティック
- `apply ... with ...` タクティック
- `injection` と `discriminate` タクティック
- タクティックを仮定に対して使う
- 帰納法の仮定を変える
- 定義の展開
- 複合的な式に関する `destruct` の使用

apply タクティック

使用法その1: 仮定や定理からゴールが直接結論づけられる時に使う

```
Theorem silly1 : forall (n m o p : nat),
     n = m  ->
     n = m.
Proof.
  intros n m eq.
(* rewrite -> eq. reflexivity. の代わりに *)
  apply eq.
Qed.
```

apply タクティック

ゴールを導く仮定が条件付の場合

```
Theorem silly2 : forall (n m o p : nat),
   n = m  ->
   (n = m -> [n;o] = [m;p]) ->
   [n;o] = [m;p].
Proof.
  intros n m o p eq1 eq2.
  apply eq2. apply eq1.
Qed.
```

- eq2 の前提となる `n = m` に遡って新しいゴールが設定される

apply タクティック

ゴールを導く仮定が条件付・全称量化されている場合

```
Theorem silly2a : forall (n m : nat),
  (n,n) = (m,m)  ->
  (forall (q r : nat),
    (q,q) = (r,r) -> [q] = [r]) ->
  [n] = [m].
Proof.
  intros n m eq1 eq2. apply eq2. apply eq1.
Qed.
```

- 全称量化された仮定 eq2 が $q := n$, $r := m$ と具体化され (n,n)=(m,m)->[n]=[m] として使われている
- 具体化された前提 (n,n) = (m,m) が新たなゴール

`apply` の使い方

より一般に，

- $Q(k)$ を示す必要がある
- 「任意の x について $P(x)$ ならば $Q(x)$」が成立することが (定理として) 既にわかっている (か，仮定されている)
- (具体化すると $P(k)$ ならば $Q(k)$ なので)，$P(k)$ を示すことにする

という推論過程に使える．
$\Longrightarrow$ 十分条件に遡る推論

apply H の挙動

仮定/定理	$H : \forall x_1, \ldots, x_m, P_1(x_1, \ldots, x_m) \to \cdots P_n(x_1, \ldots, x_m) \to Q(x_1, \ldots, x_m)$
ゴール	$Q(k_1, \ldots, k_m)$

$$\Downarrow \texttt{apply H}$$

新ゴール (n 個)	$P_1(k_1, \ldots, k_m)$ $\vdots$ $P_n(k_1, \ldots, k_m)$

symmetry タクティック

等式の左右をひっくり返す

```
Theorem silly3 : forall (n m : nat),
    n = m -> m = n.
Proof.
  intros n m H.
  (* Here we cannot use [apply] directly *)
  symmetry.
  apply H.
Qed.
```

Tactics.v

- `apply` タクティック
- `apply ... with ...` タクティック
- `injection` と `discriminate` タクティック
- タクティックを仮定に対して使う
- 帰納法の仮定を変える
- 定義の展開
- 複合的な式に関する `destruct` の使用

apply withタクティック

動機付け: 等号の推移律

```
Theorem trans_eq : forall X:Type (n m o : X),
  n = m -> m = o -> n = o.
```

をより具体的な例についての証明で使うことを考える.

```
Example trans_eq_example' :
  forall (a b c d e f : nat),
     [a;b] = [c;d] ->
     [c;d] = [e;f] ->
     [a;b] = [e;f].
Proof.
  intros a b c d e f eq1 eq2.
  apply trans_eq.   (* エラー! *)
```

何が起こったのか？

- ゴール: `[a;b] = [e;f]`
- `trans_eq :`
  `forall X (n m o: X), n = m -> m = o -> n = o`

$\Downarrow$

- 周りの状況から Coq が勝手にわかってくれること:
  - `X` := `list nat`
  - `n` := `[a;b]`
  - `o` := `[e;f]`
- 中間の `m` をどうすべきかは(ゴールだけを見ても)わからない!

Coq にヒントを与える with

```
Example trans_eq_example' :
  forall (a b c d e f : nat),
     [a;b] = [c;d] ->
     [c;d] = [e;f] ->
     [a;b] = [e;f].
Proof.
  intros a b c d e f eq1 eq2.
  apply trans_eq with (m:=[c;d]).
  apply eq1. apply eq2.
Qed.
```

- “m:=” は省略できることも
  - 定理に出てくる変数の名前を知らなくてもＯＫ

おまけ: transitivity タクティック

推移律に限っていえばこれでもよい

```
Example trans_eq_example' :
  forall (a b c d e f : nat),
     [a;b] = [c;d] ->
     [c;d] = [e;f] ->
     [a;b] = [e;f].
Proof.
  intros a b c d e f eq1 eq2.
  transitivity [c;d].
  apply eq1. apply eq2.
Qed.
```

Tactics.v

- apply タクティック
- apply ... with ... タクティック
- injection と discriminate タクティック
- タクティックを仮定に対して使う
- 帰納法の仮定を変える
- 定義の展開
- 複合的な式に関する destruct の使用

自然数について再び

自然数の性質:

場合分けの原理 (数学的帰納法の特殊ケース)

$n : nat$ ならば $n = O$ または $n = S\ n'$ なる n' が存在

コンストラクタは「単射」「1 対 1 関数」(injective)

任意の n, m について $S\ n = S\ m$ ならば $n = m$.

(対偶を取ると)

任意の n, m について $n \neq m$ ならば $S\ n \neq S\ m$.

異なるコンストラクタは等しくない

任意の n について $O \neq S\ n$ (等しいとしたら，それは矛盾)

他の帰納的定義でも同じこと

- コンストラクタの injectivity
  - 等式の左右に現れる同一コンストラクタをはがしても「等しさ」は保たれる
- 異なるコンストラクタから作られるデータは決して等しくならない

がいえる

S の単射性の証明

```
Theorem S_injective : forall (n m : nat),
  S n = S m -> n = m.
Proof.
  intros n m H1.
  assert (H2: n = pred (S n)). {reflexivity.}
  rewrite H2. rewrite H1. reflexivity.
Qed.
```

他の型のコンストラクタ (cons など) についても同様な性質の証明が可能

$$\Downarrow$$

単射性を使った証明はタクティックとして提供されている!

injection タクティック

```
Theorem S_injective' : forall (n m : nat),
  S n = S m -> n = m.
Proof.
  intros n m H.
  injection H as Hnm.
  (* 文脈に Hnm : n = m が追加される *)
  intro Hnm. apply Hnm.
Qed.
```

- (たしか) はじめての文脈を書き換えるタクティック

injection の練習 (1)

```
Theorem injection_ex1 : forall (n m o : nat),
     [n;m] = [o;o] -> [n] = [m].
Proof.
  intros n m o H.
  injection H as H1 H2.
  rewrite H1. rewrite H2. reflexivity.
Qed.
```

コンストラクタが入れ子状になっていてもまとめて分解してくれる!

- `[n;m] = cons n (cons m nil)`
- `[o;o] = cons o (cons o nil)`

injection の練習 (2)

```
Theorem injection_ex2 : forall (n m o : nat),
  [n; m] = [o; o] -> [n] = [m].
Proof.
  intros n m o H.
  injection H.
  intros H1 H2. rewrite H1. rewrite H2.
  reflexivity.
Qed.
```

as がなくても動くが，文脈に追加されていた仮定が，ゴールに含意の前提として追加される

discriminate タクティック

- 異なるコンストラクタ間の等号 (`true = false` や `0 = S x`) が仮定されている時，その矛盾を指摘し (サブ) ゴールを即座に解消する

```
Theorem eqb_0_l : forall (n m : nat),
  false = true -> n = m.
Proof. intros n m H. discriminate H. Qed.
```

```
Theorem discriminate_ex1 : forall (n : nat),
  S n = O -> 2 + 2 = 5.
Proof. intros n H. discriminate H. Qed.
```

爆発則 (principle of explosion)
「矛盾から何でも導いてよい」

もう少し複雑な例

```
Theorem eqb_0_l : forall n,
   (0 =? n) = true -> n = 0.
Proof.
  intros n.  destruct n as [| n'] eqn:E.
  - (* n = 0 *)
    intros H. reflexivity.
  - (* n = S n' *)
    (* ゴール: (0 =? S n') = true -> S n' = 0 *)
    simpl.
    intros H. discriminate H.
Qed.
```

まとめ: injection タクティック

$$\cfrac{\begin{array}{c}\vdots \\ H : c\ a_1\ \cdots\ a_n = c\ b_1\ \cdots\ b_n \\ \vdots\end{array}}{P}$$

$\Downarrow$ `injection H as H1 ... Hn.`

$$\cfrac{\begin{array}{c}\vdots \\ H_1 : a_1 = b_1 \\ \vdots \\ H_n : a_n = b_n\end{array}}{P}$$

discriminate タクティック

$$\dfrac{\begin{array}{c}\vdots \\ H : c\ a_1\ \cdots\ a_n = d\ b_1\ \cdots\ b_m \\ \vdots\end{array}}{P}$$

⇓ `discriminate H.` (c, d が違う)

仮定が矛盾しているので，このゴールは解消

寄り道: injectivity の逆

```
Theorem f_equal :
forall (A B : Type) (f: A -> B) (x y: A),
    x = y -> f x = f y.
```

```
Theorem eq_implies_succ_equal :
  forall (n m : nat),
    n = m -> S n = S m.
Proof.
  intros n m H. apply f_equal. apply H.
Qed.
```

- ゴールが等式で両辺が「ちょっとだけ違う」時に使うと，違う部分がゴールになって便利
- より一般的な f_equal というタクティックあり

Tactics.v

- `apply` タクティック
- `apply ... with ...` タクティック
- `injection` と `discriminate` タクティック
- タクティックを仮定に対して使う
- 帰納法の仮定を変える
- 定義の展開
- 複合的な式に関する `destruct` の使用

タクティックを仮定に対して使う

- `simpl in H`: 仮定 H の内容を単純化
- `symmetry in H`: 仮定 $H : a = b$ を $H : b = a$ にする
- `apply L in H`: 仮定 H に別の仮定 L を適用
  - $H : P(n)$ と $L : \forall x, P(x) \to Q(x)$ から
  - $H : Q(n)$ を導く

  推論の「方向」に注意!
- `rewrite L in H`: 仮定 L の等式を使って仮定 H を書き換え

前向き推論と後向き推論

前向き推論 (forward reasoning)

既知の事実や現在置いた仮定を組み合わせて，新しい判断を得る推論

後向き推論 (backward reasoning)

示したい判断 (ゴール) から，それを与える十分条件に遡る推論

- 演繹的・帰納的推論の区別とは直交なので注意
- Coq での推論は (帰納法の適用も含めて) 全て演繹的
- `apply` は後向き，`apply in H` は前向き

前方推論による証明例(1)

```
Theorem S_inj : forall (n m : nat) (b : bool),
     ((S n) =? (S m)) = b  ->
     n =? m = b.
Proof.
  intros n m b H. simpl in H. apply H.  Qed.
```

前方推論による証明例(2)

```
Theorem silly4 : forall (n m p q : nat),
  (n = m -> p = q) ->
  m = n ->
  q = p.
Proof.
  intros n m p q EQ H.
  symmetry in H. apply EQ in H. symmetry in H.
  apply H.  Qed.
```

Tactics.v

- `apply` タクティック
- `apply ... with ...` タクティック
- `injection` と `discriminate` タクティック
- タクティックを仮定に対して使う
- 帰納法の仮定を変える
- 定義の展開
- 複合的な式に関する `destruct` の使用

Coq での帰納法による証明の落とし穴

- Coq ではデータ型定義から自然に定まる帰納法以外の帰納法が使いづらい
  - Q:「Coq では累積帰納法は使えないの？」
  - A:「使えるけど使いにくい・工夫が必要です」

  ので，ついつい，ふつうの帰納法を使ってしまう

累積帰納法

「任意の x について $P(x)$」と以下は同値:

- 任意の x について $P(\mathbf{0})$ かつ $\cdots$ $P(x - \mathbf{1})$ ならば $P(x)$

Coq での帰納法による証明の落とし穴

- しかし，帰納法で証明したい「任意の x について $P(x)$」の P の選ばれ方 (これは Coq が選ぶ) によっては証明がうまくいかったりいかなかったりする
  - Coq による P の選び方を知る必要がある

例題

定理:「任意の自然数 n, m について $double(n) = double(m)$ ならば $n = m$」

証明: 帰納法による.

- $n = 0, m = 0$ の場合: 自明に「$double(n) = double(m)$ ならば $n = m$」
- $n = 0, m = S(m')$ の場合: そもそも $double(n) \neq double(m)$
- $n = S(n'), m = 0$ の場合: そもそも $double(n) \neq double(m)$
- $n = S(n'), m = S(m')$ の場合: (次のスライド)

例題

- $n = S(n'), m = S(m')$ の場合:
  1. $double(S(n')) = double(S(m'))$ を仮定する.
  2. この時，$double$ の定義より，
     $S(S(double(n'))) = S(S(double(m')))$ である.
  3. さらに，S の injectivity より
     $double(n') = double(m')$ である
  4. 帰納法の仮定より $n' = m'$ である
  5. これより $S(n') = S(m')$ すなわち，$n = m$ がいえた

これ，どんな帰納法？

実は自然数のペアについての帰納法!

- $P(0,0)$
- $x > 0$ ならば $P(x,0)$
- $y > 0$ ならば $P(0,y)$
- $P(x,y)$ ならば $P(x+1,y+1)$

から，

- 任意の自然数 x, y について $P(x,y)$

を示した

Coq でやってみる

```
Theorem double_injective_FAILED : forall n m,
     double n = double m ->
     n = m.
Proof.
  intros n m.  induction n as [| n'].
    ⋮
```

- ここでの $P(n)$ は
  `double n = double m -> n = m.`
- 示すべきことは
  - $P(0)$ と $P(n') \to P(S(n'))$
- m が固定されている (文脈に $m : nat$ がある) ことに注目!

なんかおかしい気もするが，突撃!

$P(0)$ を示す

```
- (* n = 0 *)
  simpl. intros eq. destruct m as [| m'].
  + (* m = 0 *) reflexivity.
  + (* m = S m' *) discriminate eq.
```

- m について場合分け．$m = S(m')$ の場合，示せそうもないゴールになるが，同時に仮定も矛盾するので discriminate で OK
- simpl は見通しをよくするため (特に必要ではない)

さらに進撃!

$\forall n', P(n') \to P(S(n'))$ を示す.

```
- (* n = S n' *)
  intros eq. destruct m as [| m'].
  + (* m = O *) discriminate eq.
  + (* m = S m' *)  apply f_equal.
```

- ※でのゴールは $\forall n', P(n') \to P(S(n'))$ そのものではなく，n', 帰納法の仮定である $P(n')$ と $P(S(n'))$ の仮定部分 `double n' = double m -> n' = m` が文脈に移っている
- m について場合わけ．$m = 0$ の場合は一撃だが…

あれ？

```
1 focused subgoals (unfocused: 0-0)
, subgoal 1 (ID 480)

  n' : nat
  m' : nat
  IHn' : double n' = double (S m') -> n' = S m'
  eq : double (S n') = double (S m')
  ============================
   n' = m'
```

- *IHn′* の結論が `n' = m'` じゃないぞ!?

何がまずかったのか

- 失敗した証明の最初で Coq に(暗黙に)伝えたこと:
  - 「n, m を何らかの自然数としよう．その n, m について $double\ n = double\ m$ ならば $n = m$ であることを n についての帰納法で示そうと思う」
- その結果のゴール:
  - $double\ \mathbf{0} = double\ m$ ならば $\mathbf{0} = m$ であること
  - 「$double\ k = double\ m$ ならば $k = m$」が「$double\ (S\ \ k) = double\ m$ ならば $S\ \ k = m$」を含意すること
    - ★ つまり「$P(k, m)$ ならば，$P(k+1, m)$」，これは示せそうもない．

ちょっと不自然だけどうまくいく証明

- $P(0,0)$
- $x>0$ ならば $P(x,0)$
- $y>0$ ならば $P(0,y)$
- $P(x,y)$ ならば $P(x+1,y+1)$

の代わりに

- 任意の y について $P(0,y)$
- 「任意の y について $P(x,y)$」ならば「任意の y について $P(x+1,y)$」

を示す(これはふつうの数学的帰納法)

日本語による証明

帰納法の P が何かを明示しよう!

定理: 任意の自然数 n, m について $\mathit{double}\ n = \mathit{double}\ m$ ならば $n = m$

証明: n についての帰納法で「任意の m について $\mathit{double}\ n = \mathit{double}\ m$ ならば $n = m$ 」を示す.

- $n = 0$ の場合: 「任意の m について $\mathit{double}\ 0 = \mathit{double}\ m$ ならば $0 = m$」を示す. m について場合わけ.
  - $m = 0$ の場合は自明
  - $m = S(m')$ の場合は, 前提 $\mathit{double}\ 0 = \mathit{double}\ m$ が不成立

- $n = S\ n'$ の場合，ただし，任意の m について $double\ n' = double\ m$ ならば $n' = m$ である，とする (帰納法の仮定).
  この時「任意の m について $double\ (S\ n') = double\ m$ ならば $(S\ n') = m$ である」を m についての場合分けで示す.
  - $m = 0$ の場合: 前提が不成立
  - $m = S\ m'$ の場合:
    - $double(S\ n') = double(S\ m')$ とする
    - 両辺を計算，injectivity を使うと $double(n') = double(m')$
    - 帰納法の仮定より m として m' をとると，$n' = m'$

Coq での証明

```
Theorem double_injective : forall n m,
     double n = double m ->
     n = m.
Proof.
  intros n. induction n as [| n'].
     ⋮
```

- m を intros しないで induction をする.
- ここでの $P(n)$ は

  `forall m, double n = double m -> n = m.`

  「その二倍が n の二倍と等しいような任意の m について, n と m は等しい」

```
- (* n = 0 *)
  simpl. intros m eq. destruct m as [| m'].
  + (* m = 0 *) reflexivity.
  + (* m = S m' *) discriminate eq.
```

- m についての場合分けの前に intros m が必要

```
- (* n = S n' *)  (* ※1 *)
  intros m eq. (* ここで m を「固定」する *)
  destruct m as [| m'].
  + (* m = O *)  discriminate eq.
  + (* m = S m' *)
    apply f_equal. (* ※2 *)
    apply IHn'. (* simpl in eq. *)
    injection eq as goal. apply goal. Qed.
```

- (※1) 失敗した時よりも一般的なこと ($\forall m$ つき) を示すことを求められている
- が，帰納法の仮定 IHn' にも $\forall m$ がついている!

(※2) の文脈とゴールは···

```
1 focused subgoals (unfocused: 0-0)
, subgoal 1 (ID 545)

  n' : nat
  IHn' : forall m : nat,
           double n' = double m -> n' = m
  m' : nat
  eq : double (S n') = double (S m')
  ============================
   n' = m'
```

- 帰納法の仮定の「任意の m」を m' で具体化してやればよい.

教訓

- Coq のせいで不自然な帰納法を使わされることがある
- n と m についての性質を n についての帰納法で証明する時，m は量化されたまま残しておくと吉な場合がある
- 「いつ残すべきか」についての確実な処方箋は「よく考えること」
  - 「$P(n, m)$ ならば $P(S(n), m)$」はいえ (そうも) ないが,「$P(n, m)$ ならば $P(S(n), S(m))$」はいえるような時は要注意

量化の順番と再量化

さきほどの定理を *m* に関する帰納法で証明しようとすると同じく失敗する.

```
Theorem double_injective : forall n m,
     double n = double m ->
     n = m.
Proof.
  intros n m. induction m as [| n'].
  (* intro なしで induction m しても同じ! *)
    ⋮
```

generalize dependent タクティック

文脈で仮定した変数を再び全称量化するタクティック

```
Theorem double_injective_take2 : forall n m,
  double n = double m -> n = m.
Proof.
  intros n m.
  generalize dependent n.
  (* ゴールが望む forall n, ... の形になる *)
  induction m as [| m'].
    ⋮
```

Tactics.v

- apply タクティック
- apply ... with ... タクティック
- injection と discriminate タクティック
- タクティックを仮定に対して使う
- 帰納法の仮定を変える
- 定義の展開
- 複合的な式に関する destruct の使用

unfold タクティック

(Definition による) 定義を展開するタクティック

```
Definition square (n:nat) := n * n.
Lemma square_mult : forall n m,
    square (n*m) = square n * square m.
Proof.
  intros n m.
  simpl.  (* 何もしてくれない *)
  unfold square.  (* square の定義の展開 *)
  rewrite mult_assoc.
  assert (H : n * m * n = n * n * m).
    rewrite mult_comm. apply mult_assoc.
  rewrite H. rewrite mult_assoc. reflexivity.
  Qed.
```

simpl について

simpl が定義の展開をしてくれる条件

定義を展開するのは，それによって場合分け (match) の計算が進む時のみ.

```
Definition foo (x: nat) := 5.
Fact silly_fact_1 : forall m,
  foo m + 1 = foo (m + 1) + 1.
Proof.
  intros m. simpl.
  (* 5 + 1 の計算途中で場合分けがあるので進む *)
  reflexivity.
Qed.
```

```
Definition bar (x: nat) :=
  match x with
  | O => 5
  | S _ => 5
 end.
Fact silly_fact_2 : forall m,
  bar m + 1 = bar (m + 1) + 1.
Proof.
  intros m. simpl.
  (* bar 中の match m with
     でひっかかるので展開前に戻る*)
```

```
unfold bar.
 (* ひっかかってるところがわかる *)
destruct m eqn:E.
- reflexivity.
- reflexivity.
```

Tactics.v

- apply タクティック
- apply ... with ... タクティック
- injection と discriminate タクティック
- タクティックを仮定に対して使う
- 帰納法の仮定を変える
- 定義の展開
- 複合的な式に関する destruct の使用

複合的な式に関する場合分け

- destruct の引数は文脈にある変数でなくてもよい
  - 実は他のタクティックも変数以外の引数が取れるものが多い
- 式一般についての場合わけが可能

```
Definition sillyfun (n : nat) : bool :=
  if n =? 3 then false
  else if n =? 5 then false
  else false.

Theorem sillyfun_false : forall (n : nat),
  sillyfun n = false.
```

- =? の結果 (n が 3 かどうか，5 かどうか) についての場合分け
  - n が 5 以下の場合を個別に，$n \geq \mathbf{6}$ の場合を帰納法で証明，という手もあるかもしれないが…

```
Proof.
  intros n. unfold sillyfun.
  destruct (n =? 3) eqn:E1.
    - (* n =? 3 = true *) reflexivity.
    - (* n =? 3 = false *)
      destruct (n =? 5) eqn:E2.
      + (* n =? 5 = true *) reflexivity.
      + (* n =? 5 = false *) reflexivity.
Qed.
```

別の例

```
Definition sillyfun1 (n : nat) : bool :=
  if n =? 3 then true
  else if n =? 5 then true
  else false.

Theorem sillyfun1_odd : forall (n : nat),
     sillyfun1 n = true ->
     oddb n = true.
```

- sillyfun1 が真を返すための必要条件は「引数が奇数」

```
Proof.
  intros n eq. unfold sillyfun1 in eq.
  (* eq : (if n =? 3 then ...) = true
     ========================
     oddb n = true                                   *)
  destruct (n =? 3).
  - (* n =? 3 = true *)
  (* eq : true = true   ←左辺の単純化の結果
     ========================
     oddb n = true                                   *)
```

- n =? 3 = true の場合と，n =? 3 = false の場合で分けたつもりなのに，肝心の n =? 3 = true が消えている!!

destruct eqn: の真の出番!

場合分けの対象の式についての情報を文脈に追加

```
Proof.
  intros n eq. unfold sillyfun1 in eq.
  (* eq : (if n =? 3 then ...) = true
     =========================
     oddb n = true                                   *)
  destruct (n =? 3) eqn:Heqe3.
  (* Heqe3 : n =? 3 = true
     eq : true = true  (* 左辺は計算済 *)
     =========================
     oddb n = true                                   *)
```

あとは，`Heqe3` から `n = 3` がわかる．

```
- (* e3 =true *)
apply eqb_true in Heqe3.
(* Heqe3 : n = 3 になる *)
rewrite -> Heqe3.   (* n = 3 を代入 *)
reflexivity.
```

- *e*3 = *false* の場合は，さらに
  `destruct (n =? 5) eqn:Heqe5.`
  で場合分け

タクティックまとめ

スライドにまとめるには多すぎるので教科書を見てください.

宿題： / 午前10:30 締切

- Exercise: `apply_exercise1` (3), `injection_ex3` (3), `discriminate_ex3` (1), `eqb_true` (2), `destruct_eqn_practice` (2)
- 解答が記入された `Tactics.v` を origin/master に push