

「計算と論理」

Software Foundations

その5

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学

November 19, 2013

今日のメニュー

- 前菜: Poly.v(データとしての関数)
 - ▶ ...
 - ▶ 関数を作る関数
 - ▶ unfold タクティック
- メイン: MoreCoq.v

関数を作る関数

- 部分適用で見たように，二引数関数は「一関数を返す関数」と考えられる
- 関数を返す関数のもっと積極的な(?)使い方を見てみよう
 - ▶ 定数関数を作る関数
 - ▶ 関数の挙動を一部変更する関数

定数関数を作る関数

```
Definition constfun {X: Type}
  (x: X) : nat->X :=
  fun (k:nat) => x.
```

```
Definition constfun' (* これでも同じ *)
  {X: Type} (x: X) (k: nat) : X := x.
```

```
Definition ftrue := constfun true.
(* a function that always returns true *)
```

```
Example constfun_example1: ftrue 0 = true.
```

```
Example constfun_example2: (constfun 5) 99 = 5.
```

関数の一部の返り値の変更

- f : 自然数を定義域とする関数
- 自然数 k と何らかのデータ x

$$f[k \mapsto x] \stackrel{\text{def}}{=} g \text{ s.t. } \begin{cases} g(k) = x \\ g(n) = f(n) \text{ (if } n \neq k) \end{cases}$$

Definition override $\{X: \text{Type}\}$

```
(f: nat->X) (k:nat) (x:X) : nat->X :=  
fun (k':nat) => if beq_nat k k' then x  
                else f k'.
```

```
Definition fmostlytrue :=  
  override (override ftrue 1 false) 3 false.
```

```
Example override_example1 :  
  fmostlytrue 0 = true.
```

```
Example override_example2 :  
  fmostlytrue 1 = false.
```

```
Example override_example3 :  
  fmostlytrue 2 = true.
```

```
Example override_example4 :  
  fmostlytrue 3 = false.
```

- この後教科書では `override` を沢山使います
- 色々性質を証明します
- そのためにもう少しタクティックを覚えましょう
- ...といっても，この講義でカバーする教科書の範囲では，もう `override` は出てこないのですが...

unfold タクティック

(Definition による) 定義を展開するタクティック

```
Theorem unfold_example : forall m n,  
  3 + n = m ->  
  plus3 n + 1 = m + 1.
```

Proof.

```
intros m n H.
```

(* 3 + n と plus3 n は定義

Definition plus3 x := (plus 3) x.

により等しいが, (見た目) は違う *)

```
unfold plus3.
```

```
rewrite -> H.
```

```
reflexivity. Qed.
```



```
Theorem override_eq :  
  forall {X:Type} x k (f:nat->X),  
    (override f k x) k = x.
```

Proof.

```
intros X x k f.
```

(* simpl. では Definition は展開されない! *)

(* compute. は展開するが,
 ゴールはもはや理解不能 *)

```
unfold override.
```

```
rewrite <- beq_nat_refl.
```

```
reflexivity.
```

Qed.

今日のメインメニュー

MoreCoq.v

- `apply` タクティック
- `apply ... with ...` タクティック
- `inversion` 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
- 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,o] = [n,p] ->  
  [n,o] = [m,p].
```

Proof.

```
  intros n m o p eq1 eq2.  
  rewrite <- eq1.  
  (* ゴールは仮定 eq1 と同じ [n,o] = [n,p] *)  
  (* rewrite -> eq2. reflexivity. の代わりに *)  
  apply eq2.
```

Qed.

apply タクティック

使用法その2: ゴールを導くための前提条件に遡る

```
Theorem silly2 : forall (n m o p : nat),  
  n = m ->  
  (forall (q r : nat), q = r -> [q,o] = [r,p])  
  [n,o] = [m,p].
```

Proof.

```
intros n m o p eq1 eq2.  
apply eq2. apply eq1.
```

Qed.

- 全称量化された仮定 $eq2$ が $q := n, r := m$ と具体化されて使われている
- 具体化された前提条件 $n = m$ が新たなゴールとなる

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 $Q(x)$ 」が成立することは (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．

⇒ 十分条件に遡っている!

apply H の挙動

仮定

$$\begin{aligned} H : \forall x_1, \dots, x_m, \\ P_1(x_1, \dots, x_m) \rightarrow \\ \dots \\ P_n(x_1, \dots, x_m) \rightarrow Q(x_1, \dots, x_m) \end{aligned}$$

ゴール

$$Q(k_1, \dots, k_m)$$

↓ apply H

新ゴール $P_1(k_1, \dots, k_m)$

(n 個)

⋮

$$P_n(k_1, \dots, k_m)$$

ゴールと apply の引数の結論のマッチング

```
Theorem silly3_firsttry : forall (n : nat),  
  true = beq_nat n 5 ->  
  beq_nat (S (S n)) 7 = true.
```

Proof.

```
  intros n H.
```

```
  simpl.
```

```
  (* Here we cannot use [apply] directly *)
```

Abort.

symmetry タクティク

等式の左右をひっくり返す

```
Theorem silly3 : forall (n : nat),  
  true = beq_nat n 5  ->  
  beq_nat (S (S n)) 7 = true.
```

Proof.

```
  intros n H.
```

```
  symmetry.
```

```
  simpl. (* 実は不要: apply は単純化を先に行う! *)
```

```
  apply H.
```

Qed.

今日のメインメニュー

MoreCoq.v

- `apply` タクティック
- `apply ... with ...` タクティック
- `inversion` 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
- 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



- Coq がわかってくれること:
 - ▶ `X := list nat`
 - ▶ `n := [a, b]`
 - ▶ `o := [e, f]`
- `m` をどうすべきかはわからない!!

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.

今日のメインメニュー

MoreCoq.v

- `apply` タクティック
- `apply ... with ...` タクティック
- `inversion` 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
- 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)

$n : \text{nat}$ ならば $n = 0$ または $n = S n'$ なる n' が存在

コンストラクタは「1対1関数」(injective)

任意の n, m について $S n = S m$ ならば $n = m$

異なるコンストラクタは異なる

任意の n について $0 \neq S n$ (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
- 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

⇒ これらを活用するのが inversion タクティク

inversion タクティック (1)

⋮

$$H : c a_1 \cdots a_n = d b_1 \cdots b_m$$

⋮

P

↓ inversion H. (c, d が同じ場合)

⋮

$$H_1 : a_1 = b_1$$

⋮

$$H_n : a_n = b_n$$

⋮

P' (P に対して H_1, \dots, H_n を使い書き換えた結果)

inversion の練習 (1)

Theorem eq_add_S : forall (n m : nat),
S n = S m -> n = m.

Theorem silly4 : forall (n m : nat),
[n] = [m] -> n = m.

Theorem silly5 : forall (n m o : nat),
[n,m] = [o,o] -> [n] = [m].

(* 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 *)

inversion (2)

$$\begin{array}{c} \vdots \\ \mathbf{H} : \mathbf{c} \mathbf{a}_1 \cdots \mathbf{a}_n = \mathbf{d} \mathbf{b}_1 \cdots \mathbf{b}_m \\ \vdots \end{array}$$

P

↓ inversion H. (**c, d が違う場合**)
仮定が矛盾しているので, このゴールは解消

inversion の練習 (2)

前提が矛盾しているので何でもいえる!

Theorem silly6 : forall (n : nat),
S n = 0 -> 2 + 2 = 5.

Theorem silly7 : forall (n m : nat),
false = true -> [n] = [m].

inversion の練習 (3)

Theorem length_snoc' :

```
forall (X : Type)
```

```
  (v : X) (l : list X) (n : nat),
```

```
  length l = n -> length (snoc l v) = S n.
```

Proof.

```
intros X v l. induction l as [| v' l'].
```

```
Case "l = []". (* 省略 *)
```

Case " $l = v' :: l'$ ".

```
intros n eq. simpl. destruct n as [| n'].
```

```
SCase "n = 0".
```

```
inversion eq.
```

```
SCase "n = S n'".
```

```
apply eq_remove_S. apply IHl'.
```

(* 仮定 eq の左辺は S (length l') に等しい *)

```
inversion eq. reflexivity.
```

Qed.

injectivity の逆

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

- 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
- (特に `inversion` とは関係ない)

今日のメインメニュー

MoreCoq.v

- `apply` タクティック
- `apply ... with ...` タクティック
- `inversion` 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
- 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** : $\forall x, P(x) \rightarrow Q(x)$ から
 - ▶ **H** : $Q(n)$ を導く

推論の「方向」に注意!

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、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 (妥当というべき?)
- `apply` は後向き, `apply in H` は前向き

例

```
Theorem S_inj : forall (n m : nat) (b : bool),  
  beq_nat (S n) (S m) = b ->  
  beq_nat n m = b.
```

Proof.

```
intros n m b H. simpl in H. apply H. Qed.
```

例

```
Theorem silly3' : forall (n : nat),  
  (beq_nat n 5 = true ->  
    beq_nat (S (S n)) 7 = true) ->  
  true = beq_nat n 5 ->  
    true = beq_nat (S (S n)) 7.
```

Proof.

```
intros n eq H.  
symmetry in H. apply eq in H. symmetry in H.  
apply H. Qed.
```

今日のメインメニュー

MoreCoq.v

- `apply` タクティック
- `apply ... with ...` タクティック
- `inversion` 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
- 複合的な式に関する `destruct` の使用

帰納法による証明の落とし穴

- 帰納法は「任意の x について、 $P(x)$ 」の形の判断を証明する技法
- P の選び方によって証明がうまくいったりいかなかったりする
- (既に似た現象に遭遇した人もいるでしょう)

うまくいかない証明の例 (1/3)

```
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
 $\text{double } n = \text{double } m \rightarrow n = m.$
- 示すべきことは
 - ▶ $P(0)$
 - ▶ $\forall n', P(n') \rightarrow P(S(n'))$

うまくいかない証明の例 (2/3)

P(0) を示す

Case "n = 0".

```
simpl. intros eq. destruct m as [| m'].
```

```
SCase "m = 0". reflexivity.
```

```
SCase "m = S m'". inversion 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 `inversion` で OK
- `simpl` は見通しをよくするため (特に必要ではない)

うまくいかない証明の例 (3/3)

$\forall n', P(n') \rightarrow P(S(n'))$ を示す .

Case " $n = S n'$ ".

```
intros eq. destruct m as [| m'].
```

```
SCase "m = 0". inversion eq.
```

```
SCase "m = S m'". apply f_equal.
```

- でのゴールは $\forall n', P(n') \rightarrow P(S(n'))$ 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...

1 subgoals, subgoal 1 (ID 580)

SCase := "m = S m'" : String.string

Case := "n = S n'" : String.string

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!?

何がまずかったのか

- P 中の m は文脈にある m , すなわち (何か正体はわからないが) 特定の m を参照している
- 失敗した証明の最初で Coq 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 $\text{double } n = \text{double } m$ 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:
 - ▶ $\text{double } 0 = \text{double } m$ ならば $0 = m$ であること
 - ▶ 「 $\text{double } k = \text{double } m$ ならば $k = m$ 」が
「 $\text{double } (S k) = \text{double } m$ ならば $S k = m$ 」を
含意すること
 - ★ このゴールはおかしい (示せそうもない)!

何がおかしいの？

正体はわからないが特定の m について，

「 $\text{double } k = \text{double } m$ ならば $k = m$ 」

であることがわかれば，

「 $\text{double } (S\ k) = \text{double } m$ ならば $S\ k = m$ 」
は証明できる

という主張!?($m = 5$ だとして考えてみよ!)

より「強い」主張を選ぶ

```
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 .

Case "n = 0".

```
  simpl. intros m eq. destruct m as [| m'].
```

```
  SCase "m = 0". reflexivity.
```

```
  SCase "m = S m'". inversion eq.
```

- m についての場合分けの前に `intros m` が必要

```

Case "n = S n'". (* 1 *)
  intros m eq. (* ここで m を「固定」する *)
  destruct m as [| m'].
  SCase "m = 0". inversion eq.
  SCase "m = S m'".
    apply f_equal. (* 2 *)
    apply IHn'. (* simpl in eq. *)
    inversion eq. reflexivity. 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 subgoals, subgoal 1 (ID 663)

```
SCase := "m = S m'" : String.string
Case  := "n = S n'" : String.string
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

- 帰納法を使う時にゴールが過度に具体的になっていないか注意せよ
- 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必要があることがある
- 「いつ残すべきか」についての処方箋は残念ながらないです...

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'].
```

```
  ...
```

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

$\text{double } n = \text{double } m$ ならば $n = m$ である

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

- $n = 0$ の場合: 「任意の m について $\text{double } 0 = \text{double } m$ ならば $0 = m$ である」を示す .

⋮

- $n = S n'$ の場合 , ただし , 任意の m について $\text{double } n' = \text{double } m$ ならば $n' = m$ である , とする .
この時 「任意の m について $\text{double } (S n') = \text{double } m$ ならば $(S n') = m$ である」を示す .

今日のメインメニュー

MoreCoq.v

- `apply` タクティック
- `apply ... with ...` タクティック
- `inversion` 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
- 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)
- 式一般についての場合分けが可能

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

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

- `beq_nat` 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, $n \geq 6$ 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 (beq_nat n 3).
```

```
Case "beq_nat n 3 = true". reflexivity.
```

```
Case "beq_nat n 3 = false".
```

```
  destruct (beq_nat n 5).
```

```
    SCase "beq_nat n 5 = true". reflexivity.
```

```
    SCase "beq_nat n 5 = false". reflexivity.
```

Qed.

別の例

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

```
oddb n = true *)
```

```
destruct (beq_nat n 3).
```

```
Case "beq_nat n 3 = true".
```

```
(* eq : true = true      左辺の単純化の結果  
=====
```

```
oddb n = true *)
```

- $\text{beq_nat } n \ 3 = \text{true}$ の場合と,
 $\text{beq_nat } n \ 3 = \text{false}$ の場合で分けたつもりなの
に, 肝心の $\text{beq_nat } n \ 3 = \text{true}$ が消えている!!

destruct eqn: タクティック

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 beq_nat n 3 then ...) = true  
=====  
oddb n = true *)  
destruct (beq_nat n 3) eqn:Heqe3.  
(* Heqe3 : beq_nat 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 .

Case "e3 =true".

apply beq_nat_eq in Heqe3.

(* Heqe3 : $n = 3$ になる *)

rewrite -> Heqe3. (* $n = 3$ を代入 *)

reflexivity.

- **e3 = false** の場合は , さらに

destruct beq_nat n 5 eqn:Heqe5.

で場合分け

宿題：12/3 午前10:30 締切

- Exercise: `silly_ex` (2), `sillyex1` (1), `sillyex2` (2), `plus_n_n_injective` (3), `beq_nat_true` (2), `gen_dep_practice` (3), `override_shadow` (1), `destruct_eqn_practice` (2)
- 解答を書き込んだ **MoreCoq.v** までのファイル全てをオンライン提出システムを通じて提出
- 以下をコメント欄に明記:
 - ▶ 講義・演習に関する質問，わかりにくいと感じたこと，その他気になること．（「特になし」はダメです．）
 - ▶ 友達に教えてもらったなら、その人の名前，他の資料（web など）を参考にした場合，その情報源（URL など）．