「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 13, 2012

コメント欄より

Q: 参考書はありませんか？

この講義の内容は

- (静的型付) 関数型プログラミング
- 数理論理学
- 型理論
- 証明支援系ソフトウェアを使ったプログラム検証

を適度にミックスして構成されています．それぞれについていくつか参考書を挙げます．
色々なデータに対する帰納法を使った証明技法について書かれた (日本語の) 本があると役に立つのですが，聞いたことがないです．

関数型プログラミング

- H. Abelson, G. J. Sussman, J. Sussman. Structure and Interpretation of Computer Programs. 2nd ed., The MIT Press, 1996. **言わずと知れた「アルゴリズムとデータ構造入門」「プログラミング言語」の教科書．**(**タイトルを略して** SICP **とも**)
- M. Felleisen, R. Findler, M. Flatt, S. Krishnamurthi. How to Design Programs: An Introduction to Programming and Computing. The MIT Press, 2001. SICP **と同様** Scheme (**の拡張**) **を使った** (**もう少し優しい**) **プログラミング入門書．**
- **浅井 健一**. **プログラミングの基礎**. Computer Science Library **シリーズ**, **サイエンス社**, 2007. **プログラミング言語は** (**静的型付言語である**) OCaml **を使っている．** How to Design ... **の方法論を参考にしている．**

数理論理学・型理論

この内容はまだ講義では薄いです (というか影に隠れています) が…

- **萩谷 昌己, 西崎 真也. 論理と計算のしくみ. 岩波書店, 2007.**
  いくつかの形式論理と λ 計算という計算のモデル，そして型理論のひとつとして OCaml など静的型付言語のモデルである型付 λ 計算と形式論理との関係を学べる．

証明支援系ソフトウェア

- Y. Bertot, P. Castéran. Interactive Theorem Proving And Program Development: Coq'art: The Calculus Of Inductive Constructions. Springer-Verlag, 2004. Coq **の入門書．この講義よりも論理に主眼が置かれている．**
- T. Nipkow, L. Paulson, M. Wenzel. Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Springer-Verlag, 2002. Isabelle/HOL **という証明支援系の入門書．色々なデータに対する帰納法を使ってプログラムの正しさを証明する，という意味ではこの教科書に近い．**

  Nipkow **さんのホームページによると，出版された本の内容は古くなってしまったので**

  `http://isabelle.in.tum.de/doc/tutorial.pdf` **を読め，とのことです．**

Q: Theorem, Lemma の使い分けは？

- Coq にとってはどれも同じことをするコマンド
- 通常，数学では証明した人の価値判断によってどう呼ぶかが変わってきます
  - 定理: 一番えらい，これが証明したかった．
  - 系: 特殊ケースだったりして定理からすぐに出てくるもの
  - 補題: 定理を証明するために仕方なく (？) 証明したもの．場合によっては補題なのに有名なものもある (「Zorn の補題」「米田の補題」)
  - 命題: 補題と定理の間くらいのえらさ？

Q: 前のファイルに書いた定義がうまく働かないが？

- `Require Export` が読むのは .vo ファイル
- `admit` で定義されていた関数を書き換えた場合は.v ファイルから .vo を再生成する必要あり
  - linux なら .v ファイルがあるところで (linux の) make コマンドを実行
  - もしくは Lists.v 冒頭にある方法を参考にしてください

今日のメニュー

Poly.v **後半**

- **データとしての関数**
  - 高階関数
  - 部分適用
  - より道: カリー化 (**省略**)
  - 高階関数カタログ
  - **関数を作る関数**
- **もっと** Coq **について**

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

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

```
Definition override {X: 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魔法の呪文 (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…

今日のメニュー

Poly.v 後半

- データとしての関数
- もっと Coq について
  - apply タクティック
  - unfold タクティック
  - inversion タクティック
  - 帰納法の仮定について
  - タクティックを仮定に対して使う
  - 複合的な式に関する destruct の使用
  - remember タクティック
  - apply ... with ... タクティック

apply タクティック

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 タクティックの使用例その2

```
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**
- **仮定の前提条件** 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．

apply H の挙動

仮定 $H : \forall x_1, \ldots, x_m,$
$P_1(x_1, \ldots, x_m) \to$
$\ldots$
$P_n(x_1, \ldots, x_m) \to Q(x_1, \ldots, x_m)$

ゴール $Q(k_1, \ldots, k_m)$

$\Downarrow$ apply H

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

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 *)
Admitted.
```

symmetryタクティック

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 H.
Qed.
```

unfoldタクティック

定義をわざと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. では展開されない! *)
  unfold override.
  rewrite <- beq_nat_refl.
  reflexivity.
Qed.
```

今日のメニュー

Poly.v 後半

- データとしての関数
- もっと Coq について
  - `apply` タクティック
  - `unfold` タクティック
  - `inversion` タクティック
  - 帰納法の仮定について
  - タクティックを仮定に対して使う
  - 複合的な式に関する `destruct` の使用
  - `remember` タクティック
  - `apply ... with ...` タクティック

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)

$n : \mathsf{nat}$ ならば $n = \mathsf{O}$ または $n = \mathsf{S}\ n'$ 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 $\mathsf{S}\ n = \mathsf{S}\ m$ 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 $\mathsf{O} \neq \mathsf{S}\ n$ (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

$\Longrightarrow$ これらを活用するのが `inversion` タクティック

inversion タクティック (1)

$$\vdots$$

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

$$\vdots$$

$\mathbf{P}$

$\Downarrow$ `inversion H.` ($\mathbf{c}$, $\mathbf{d}$ が同じ場合)

$$\vdots$$

$$\mathbf{H}_1 : \mathbf{a}_1 = \mathbf{b}_1$$

$$\vdots$$

$$\mathbf{H}_n : \mathbf{a}_n = \mathbf{b}_n$$

$$\vdots$$

$\mathbf{P}'$ ($\mathbf{P}$ に対して $\mathbf{H}_1, \ldots, \mathbf{H}_n$ 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].
```

inversion (2)

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

$\Downarrow$ `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)

```
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.
```

今日のメニュー

Poly.v 後半

- データとしての関数
- もっと Coq について
  - `apply` タクティック
  - `unfold` タクティック
  - `inversion` タクティック
  - 帰納法の仮定について
  - タクティックを仮定に対して使う
  - 複合的な式に関する `destruct` の使用
  - `remember` タクティック
  - `apply ... with ...` タクティック

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 beq_nat_eq_FAILED : forall n m,
  true = beq_nat n m -> n = m.
Proof.
  intros n m H. induction n as [| n'].
```

- **ここでの P(n) は** `true = beq_nat n m -> 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". (* ※ *)
  destruct m as [| m'].
  SCase "m = 0". reflexivity.
  SCase "m = S m'". inversion H.
```

- **※でのゴールは P(0) のそのものではなく，「ならば」の仮定が文脈に移されたものになっている**
- **m について場合分け．$m > 0$ とすると，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
- **教科書では** `simpl in H` **があるが，これは直後に習うもの．**

うまくいかない証明の例 (3/3)

$\forall n', P(n') \to P(S(n'))$ を示す．

```
Case "n = S n'".  (* ※ *)
  destruct m as [| m'].
  SCase "m = 0". inversion H.
  SCase "m = S m'".
    apply eq_remove_S.
```

- **※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 `true = beq_nat (S 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 subgoal

  SCase := "m = S m'" : String.string
  Case := "n = S n'" : String.string
  n' : nat
  m' : nat
  H : true = beq_nat (S n') (S m')
  IHn' : true = beq_nat n' (S m') -> n' = S m'
  ============================
   n' = m'
```

- **IHn′ は，最初に存在を仮定した (具体的には何かわからないが) 特定の m についての性質を述べている．**
- **帰納法の仮定が弱い**

「強い」帰納法の仮定を選ぶ

```
Theorem beq_nat_eq : forall n m,
  true = beq_nat n 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, true = beq_nat n m -> n = m.
```

```
Case "n = 0".  (* ※ *)
  intros m. destruct m as [| m'].
  SCase "m = 0". reflexivity.
  SCase "m = S m'".
    intros contra. inversion contra.
```

- **※でのゴールは P(0) のそのもの.**
- **場合分けの前に** `intros m` **が必要**
- **矛盾した仮定なので** contra(diction) **という名前**

```
Case "n = S n'".    (* ※ *)
  intros m. destruct m as [| m'].
  SCase "m = 0".
    intros contra. inversion contra.
  SCase "m = S m'".
```

- **※でのゴールは P(S(n′)) のそのもの，帰納法の仮定 IHn′ には `forall m` がついている!**

ここでの文脈とゴールは…

```
1 subgoal

  SCase := "m = S m'" : String.string
  Case := "n = S n'" : String.string
  n' : nat
  IHn' : forall m : nat,
           true = beq_nat n' m -> n' = m
  m' : nat
  ============================
   true = beq_nat (S n') (S m') -> S n' = S m'
```

- **IHn′ は文脈の m (`forall m` の `m` とは別物!) については何も述べておらず，場合分けの影響を受けない**
- **IHn′ に対し m′ の場合を考えればうまくいきそう!**

```
SCase "m = S m'". simpl. intros H.
  apply eq_remove_S.  (* ※ *)
  apply IHn'. apply H. Qed.
```

- **※ではゴールが $n' = m'$**
- `apply IHn'.` **で自動的に $n = m'$ となる**

induction タクティックの挙動

ゴールが P(x) で induction x とすると，

- x に言及している仮定 $Q_1(x), \ldots, Q_n(x)$ が文脈にあった場合，それらを集めた

$$Q_1(x) \to \cdots Q_n(x) \to P(x)$$

もとに場合分けしたゴール・帰納法の仮定を設定

- x に言及している仮定を集めないと正しい推論にならない
  - Basics.v の mult_comm でこれが原因で嵌った人あり
- x とは関係のない仮定はそのまま．
  - intros しすぎると嵌ることあり

今日のメニュー

Poly.v 後半

- データとしての関数
- もっと Coq について
  - `apply` タクティック
  - `unfold` タクティック
  - `inversion` タクティック
  - 帰納法の仮定について
  - タクティックを仮定に対して使う
  - 複合的な式に関する `destruct` の使用
  - `remember` タクティック
  - `apply ... with ...` タクティック

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 : ∀x, P(x) → 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 (妥当というべき？)

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.
```

今日のメニュー

Poly.v 後半

- データとしての関数
- もっと Coq について
  - apply タクティック
  - unfold タクティック
  - inversion タクティック
  - 帰納法の仮定について
  - タクティックを仮定に対して使う
  - 複合的な式に関する destruct の使用
  - remember タクティック
  - apply ... with ... タクティック

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わけが可能

```
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                          *)
```

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

remember タクティック

新しい変数とそれが何かと等しいことを文脈に記録

```
Proof.
  intros n eq. unfold sillyfun1 in eq.
  (* eq : (if beq_nat n 3 then ...) = true
     ============================
     oddb n = true                          *)
  remember (beq_nat n 3) as e3.
  (* eq : ...
     e3 : bool
     Heqe3 : e3 = beq_nat n 3
     ============================
     oddb n = true                          *)
```

ここで e3 についての場合分けをする

```
destruct e3.
Case "e3 = true".
(* eq : true = true
   e3 : bool   ← 消えているはず
   Heqe3 : true = beq_nat n 3
   ============================
   oddb n = true                                    *)
```

あとは，Heqe3 から n = 3 がわかる．

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

remember を使った証明のポイント

- destruct の場合分けは対象の式を直接書き換えてしまうので，**その式が含んでいた変数との関係性が失なわれる**ことがある
- 「〜と等しい変数 x」 を remember で導入して
- x について destruct による場合分けをする
- 元の式は書き換え対象にならないので関係性も残る!

今日のメニュー

Poly.v 後半

- データとしての関数
- もっと Coq について
  - apply タクティック
  - unfold タクティック
  - inversion タクティック
  - 帰納法の仮定について
  - タクティックを仮定に対して使う
  - 複合的な式に関する destruct の使用
  - remember タクティック
  - apply ... with ... タクティック

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, n = m -> m = o -> n = o`

- Coq **がわかってくれること**:
  - `X` := `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.
```

宿題： 11/27 午前10:00 締切

- Exercise: `override_example` (1) (**ソースにコメントとして定理の意味を書くこと**), `apply_exercise1` (3) , `override_neq` (2) `sillyex1` (1), `sillyex2` (2), `apply_exercise2` (3), `plus_n_n_injective` (3),
- **解答を書き込んだ `Poly.v` をまるごと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)．**