「計算と論理」
Software Foundations
その8

五十嵐 淳
igarashi@kuis.kyoto-u.ac.jp

京都大学

December 11, 2012

今日のメニュー

Prop.v

- 帰納的に定義される命題
- 証明オブジェクト
  - 証明スクリプトと証明オブジェクト
  - 含意 (「ならば」) と関数
  - 証明オブジェクトに関する帰納法
  - 偶数性
  - 証拠についての場合わけ・inversion
- 命題を対象とするプログラミング
- 帰納法の原理について

復習: 美しい数の定義

推論規則による「美しさ」の定義:

$$\frac{}{0 \text{ is beautiful}} \quad (\textsc{b}0)$$

$$\frac{}{3 \text{ is beautiful}} \quad (\textsc{b}3)$$

$$\frac{}{5 \text{ is beautiful}} \quad (\textsc{b}5)$$

$$\frac{\mathbf{n} \text{ is beautiful} \qquad \mathbf{m} \text{ is beautiful}}{\mathbf{n} + \mathbf{m} \text{ is beautiful}} \quad (\textsc{bsum})$$

復習: 美しい数の定義

```
Inductive beautiful : nat -> Prop :=
  b_0   : beautiful 0
| b_3   : beautiful 3
| b_5   : beautiful 5
| b_sum : forall n m,
            beautiful n ->
            beautiful m ->
            beautiful (n+m).
```

証明オブジェクトに関する帰納法

- `Inductive` による型定義
  →その型のデータに関する帰納法
  - 「任意の自然数 `n` について…」の証明
- `Inductive` による命題定義
  →その命題の証明に関する帰納法 (？)
  - 「`beautiful n` ならば…」の証明

帰納的定義されるデータ vs 証明オブジェクト (1/2)

自然数 n について，以下のいずれかがいえる

- **n = O**
- **n = S n0 (n0 は n より「小さい」自然数)**

⇓

自然数に関する帰納法

「任意の n について P(n)」 と

- **P(O) かつ**
- **任意の n0 について P(n0) ならば P(S n0)**

は同値

帰納的定義されるデータ vs 証明オブジェクト (2/2)

`beautiful` **n の証明オブジェクト E について，以下のいずれかがいえる**:

- **E =** `b_0` **かつ n =** `0`
- **E =** `b_3` **かつ n =** `3`
- **E =** `b_5` **かつ n =** `5`
- **E =** `b_sum` **n1 n2 E1 E2 かつ**
  - **n = n1+n2 かつ**
  - **E1 は** `beautiful` **n1 の (E より小さい) 証明オブジェクト，かつ，**
  - **E2 は** `beautiful` **n2 の (E より小さい) 証明オブジェクト**

beautiful n の証明オブジェクトに関する帰納法，または美しい数に関する帰納法

「任意の beautiful n なる n について P(n)」つまり「任意の n について beautiful n ならば P(n)」と以下は同値

- P(0) かつ
- P(3) かつ
- P(5) かつ
- 任意の n1, n2 について P(n1) かつ P(n2) ならば P(n1 + n2)

証明オブジェクトの形に関する場合分けとの対応!

例: 帰納的定義される命題の同値性証明(1)

```
Inductive gorgeous : nat -> Prop :=
  g_0 : gorgeous 0
| g_plus3 :
   forall n, gorgeous n -> gorgeous (3+n)
| g_plus5 :
   forall n, gorgeous n -> gorgeous (5+n).
```

明らかに(?)，自然数がゴージャスであることと美しいことは同値

$\Longrightarrow$ **証明してみよう!**

- **その前に…**

ゴージャス数の証明オブジェクト

gorgeous n の証明オブジェクト E について，以下のいずれかがいえる:

- E = g_0 かつ n = 0
- E = g_plus3 n′ E1 かつ n = 3 + n′ かつ E1 は gorgeous n′ の (E より小さい) 証明オブジェクト
- E = g_plus5 n′ E1 かつ n = 5 + n′ かつ E1 は gorgeous n′ の (E より小さい) 証明オブジェクト

ゴージャスな数についての帰納法

gorgeous n の証明オブジェクトに関する帰納法，または，ゴージャスな数に関する帰納法

「任意のゴージャスな数 n について P(n)」つまり「任意の n について gorgeous n ならば P(n)」と以下は同値

- P(0) かつ
- 任意の n' について P(n') ならば P($3 + n'$)
- 任意の n' について P(n') ならば P($5 + n'$)

ゴージャスな自然数は美しい!

```
Theorem gorgeous__beautiful : forall n,
  gorgeous n -> beautiful n.
Proof.
   intros n H.
   induction H as [| n'| n'].
     (* 証明オブジェクトに関する帰納法! *)
   Case "g_0: n=0".  apply b_0.
   Case "g_plus3: n=3+n'".
     apply b_sum. apply b_3. apply IHgorgeous.
   Case "g_plus5: n=5+n'".
     apply b_sum. apply b_5. apply IHgorgeous.
Qed.
```

ゴージャス数は和について閉じている

```
Theorem gorgeous_sum : forall n m,
  gorgeous n -> gorgeous m -> gorgeous (n + m).
Proof.
  intros n m H1 H2.
  induction H1 as [| n'| n'].
  Case "n=0".     (* Show gorgeous (0+m) *)

  Case "n=3+n'". (* Show gorgeous (3+n'+m) *)


  Case "n=5+n'". (* Show gorgeous (5+n'+m) *)
    ...
Qed.
```

美しい自然数はゴージャス

```
Theorem beautiful__gorgeous : forall n,
  beautiful n -> gorgeous n.
Proof.
  intros n H.  induction H as [| | | n' m'].
    (* beautiful n についての帰納法 *)
  Case "n=0".
  Case "n=3".
  Case "n=5".
  Case "n = n' + m'".
    (* 帰納法の仮定: gorgeous n', gorgeous m' *)
Qed.
```

今日のメニュー

Prop.v

- 帰納的に定義される命題
- 証明オブジェクト
  - 証明スクリプトと証明オブジェクト
  - 含意 (「ならば」) と関数
  - 証明オブジェクトに関する帰納法
  - 偶数性
    - ★ 計算による定義と帰納的定義
  - 証拠についての場合わけ・inversion
- 命題を対象とするプログラミング
- 帰納法の原理について

ふたつの偶数性の定義

- **偶数性判定関数を使った定義**

```
Definition even (n:nat) : Prop :=
  evenb n = true.
```

  - **even n** $\overset{\text{def}}{\Longleftrightarrow}$ **evenb n = true**

- **帰納的命題定義を使った定義**

```
Inductive ev : nat -> Prop :=
| ev_0 : ev 0  (* 0 は偶数 *)
| ev_SS : forall n:nat, ev n -> ev (S (S n)).
    (* n が偶数ならば S (S n) もそう *)
```

判定関数による定義 vs 帰納的命題定義

- 偶数性については，どちらも同じくらいわかりやすい・扱いやすい
- 一般には帰納的定義の方が扱いやすいことが多い
  - 判定関数が書けない場合すらある!
    - `beautiful` かどうかの判定関数？

今日のメニュー

Prop.v

- 帰納的に定義される命題
- **証明オブジェクト**
  - 証明スクリプトと証明オブジェクト
  - 含意 (「ならば」) と関数
  - 証明オブジェクトに関する帰納法
  - 偶数性
  - **証拠についての場合わけ**・inversion
- **命題を対象とするプログラミング**
- **帰納法の原理について**

証拠についての inversion

偶数性の証拠を使った推論:

- **帰納法** (induction)
- **(素朴な) 場合わけ** (destruct)
- **証拠を「遡る」** (inversion)

例題:

```
Theorem SSev_ev : forall n,
  ev (S (S n)) -> ev n.
```

素朴な destruct による失敗例

```
Theorem SSev_ev_firsttry : forall n,
  ev (S (S n)) -> ev n.
Proof.
  intros n E.
  destruct E as [| n' E'].
    (* The goal is still "ev n" *)
```

- **n = 0 なはずなのに…**
- remember **を使えばよい?**

destruct + remember

```
Theorem SSev_ev_secondtry : forall n,
  ev (S (S n)) -> ev n.
Proof.
  intros n E. remember (S (S n)) as n2.
  destruct E as [| n' E'].
  Case "n2 = 0". inversion Heqn2.
  Case "n2 = S (S n')".
    inversion Heqn2. rewrite <- H0. apply E'.
Qed.
```

証拠に対する inversion を使った簡潔な証明

```
Theorem SSev__even : forall n,
  ev (S (S n)) -> ev n.
Proof.
  intros n E.
  inversion E as [| n' E'].
    apply E'.
Qed.
```

証拠に対する inversion

文脈で H : I (I は帰納的に定義された命題) とする時，inversion H は:

- コンストラクタ (導出規則) 毎に場合わけ
  - ev_0, ev_SS の場合
- 各場合での前提条件…
  - …を文脈に追加
    - ev_SS の場合の前提 ev n が追加
  - …が矛盾している場合は場合そのものの除去
    - ev_0 の場合 (S (S n) = 0 はありえない)

を一気に行う

今日のメニュー

Prop.v

- 帰納的に定義される命題
- 証明オブジェクト
- **命題を対象とするプログラミング**
- **帰納法の原理について**

命題を対象とするプログラミング

Coq では「命題」も「式 (プログラム)」「証明オブジェクト」と同じ言語で書かれた表現 (Prop 型の式)

```
Check (2 + 2 = 4).
Check (beautiful 8).
Check (2 + 2 = 5).
  (* 証明ができるかどうかと命題であることは別! *)
```

命題に名前をつける:

```
Definition plus_fact : Prop  :=  2 + 2 = 4.
Theorem plus_fact_is_true : plus_fact.
Proof. reflexivity.  Qed.
```

命題(Prop型の式)の作り方

- Inductive
- **等号**: = **の両側に同じ型の式を並べる**
- **含意**: **命題ふたつを** -> **で結ぶ**
- **全称量化**: **命題に** forall **をつける**

パラメータ化された命題

- 例えば `even` の型 `nat -> Prop` の読み方
  - 自然数から命題を返す**関数**
  - 自然数で添字付けされた命題の**族** (family)
  - 自然数についての**性質** (述語)
- パラメータ化された命題定義:

```
Definition between (n m o: nat) : Prop :=
  andb (ble_nat n o) (ble_nat o m) = true.
Definition teen : nat->Prop := between 13 19.
  (* 部分適用! *)
```

高階命題プログラミング

述語 P を与えると命題を返す関数:

- **「0 で P が成立する」**

```
Definition
  true_for_zero (P:nat->Prop) : Prop := P 0.
```

- **「任意の自然数 n について P が成立する」**

```
Definition
  true_for_all_numbers
     (P:nat->Prop) : Prop := forall n, P n.
```

- **「P が n′ で成立するならば S n′ でも成立する」**

```
Definition
  preserved_by_S (P:nat->Prop) : Prop :=
    forall n', P n' -> P (S n').
```

様々な全称量化された命題

- **データに関する量化**
  - `forall (n:nat), beq_nat n n = true`
  - `forall (f:nat->nat), map f [] = []`
- **型に関する量化**
  - `forall (X:Type) (l:list X), rev(rev l) = l`
- **命題に関する量化**
  - `forall (P Q:Prop), P -> (P -> Q) -> Q`
  - `forall (P:nat->Prop) (n:nat),P n -> P(n+0)`

量化と抽象化

量化 (forall) は命題や型が対象

- `forall (n:nat), beq_nat n n = true`
- `forall (X:Type), list X -> list X`

関数抽象 (fun) は計算式が対象

- `fun (n:nat) => S n`
- `fun (n:nat) => beq_nat n n`
- `fun (X:Type) (l:list X) => l ++ []`
- `fun (P Q:Prop) (H1:P->Q) (H2:P) => H1 H2`

Coq の世界(1/3)

```
                型                          プログラム
Type -+- nat                  --- O, S O, ...
      |
      +- bool                 --- true, false
      |
      +- nat -> nat           --- fun n => S (S n)
      |
      +- (nat -> nat) -> nat
                              --- fun f => f (f O)
```

Coq の世界(2/3)

	命題	証明オブジェクト
Prop	1 + 1 = 2	eq_refl 2
	even n	ev_0, ev_SS 0 ev_0, ...
	forall (n m:nat), n + m = m + n	fun n m => ...
	forall (f:nat->nat), ...	
	forall n, even n -> even (n+4)	fun n (H:even n) => ...
	1 + 2 = 5	(**なし**)

Coq の世界(3/3)

Prop も型の一種!

```
Type -+- ...
      |
      +- Prop -+- ...
      |
      +- nat -> Prop
      |          --- fun n => beq_nat n n = true
      +- Prop -> Prop --- fun P => P -> P
```

Coq の世界(3/3)

Prop も型の一種!

```
Type -+- ...
      |
      +- Prop -+- ...
      |        |
      |        +- forall (P:Prop), P -> P
      |        |                --- fun P (H:P) => H
      |        +- forall (P:nat->Prop), P 0
      |                         --- (なし)
      +- nat -> Prop
      |         --- fun n => beq_nat n n = true
      +- Prop -> Prop --- fun P => P -> P
```

宿題： 12/19 午前10:00 締切

- Exercise: `gorgeous_sum` (2), `beautiful__gorgeous` (3), `double_even` (1), `ev__even` (1), `ev_sum` (2), `inversion_practice` (1), `ev_ev__ev` (3)
- **解答を書き込んだ `Prop.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)．**

今日のメニュー

Prop.v

- 帰納的に定義される命題
- 証明オブジェクト
- 命題を対象とするプログラミング
- **帰納法の原理について**
  - **帰納的定義された型のための帰納法**
  - **帰納法の仮定**

帰納的定義された型のための帰納法

- Inductive による型 t の定義

$\Longrightarrow$ **帰納法の原理**に対応する命題の証明オブジェクトとなる定数 t_ind が生成される

```
Check nat_ind.
nat_ind
     : forall P : nat -> Prop,
         P 0  ->
         (forall n : nat, P n -> P (S n))  ->
         forall n : nat, P n
```

- 任意の自然数上の述語 P について適用可能

数学的帰納法 (再掲)

P(n) を自然数 n の性質について述べた命題とする

数学的帰納法の原理

「任意の自然数 n について P(n)」は以下と同値

- P(0) かつ
- 任意の自然数 n′ について P(n′) ならば P(S n′)

induction を使った証明

```
Theorem mult_0_r' : forall n:nat,
  n * 0 = 0.
Proof.
  intros n.  induction n as [| n'].
  Case "n=0".  reflexivity.
  Case "n=S n'".  simpl.  apply IHn'.
Qed.
```

induction を使わない別証明

```
Proof.
  apply nat_ind.
  Case "O". reflexivity.
  Case "S". simpl. intros n IHn.
            rewrite -> IHn. reflexivity.
Qed.
```

- nat_ind の形とあわせるため，intros n. がない
- 帰納法の仮定は自分で intros する
- サブゴールの変数名のチョイスは Coq 任せ
  - Case "S" で n が衝突

帰納法の原理の一般形

```
Inductive t : Type :=
  c1 : ... -> t
  :
| cn : ... -> t
```

```
t_ind :
    forall P : t -> Prop,
          ... case for c1 ... ->
                 ...          ->
          ... case for cn ... ->
          forall n : t, P n
```

例(1)

```
Inductive yesno : Type :=
  | yes : yesno
  | no : yesno.

Check yesno_ind.
```

例(2)

```
Inductive natlist : Type :=
  | nnil : natlist
  | ncons : nat -> natlist -> natlist.

Check natlist_ind.
```

帰納法の原理の一般形

- 型宣言のコンストラクタ $\leftrightarrow$ 帰納法の場合わけ
- コンストラクタ c の引数の型が $a_1, \ldots, a_n$ とする.
- a_i のいくつかは t (今定義しようとしている型) かもしれない
- 各場合に対応するゴール命題:

  任意の $x_1 : a_1, \ldots, x_n : a_n$ について，もし P が各 $x_i : t$ について成立するならば，P が c x_1 $\cdots$ x_n についても成立する

  - 青い部分が帰納法の仮定

多相的データ型に対する帰納法の原理

```
Inductive list (X:Type) : Type :=
  | nil : list X
  | cons : X -> list X -> list X.

Check list_ind.
```

- **基本的には** `natlist` **と同様**
- **定義全体が** `X` **でパラメータ化されている**
- `list_ind` **は型** T **を与えると**
  T を要素とするリストのための帰納法の原理
  を返す (多相) 関数