

「計算と論理」

Software Foundations

その7

五十嵐 淳

`cal16@fos.kuis.kyoto-u.ac.jp`

<http://www.fos.kuis.kyoto-u.ac.jp/~igarashi/class/cal/>

December 20, 2016

IndProp.v

- 帰納的に定義される命題
- 証明オブジェクトを証明で使う
 - ▶ 導出についての inversion
 - ▶ 導出についての帰納法
- (帰納的に定義される) 関係
- ケーススタディ: 正則表現
- (ケーススタディ: reflection の改良)

命題を返す関数を使った偶数性の定義 (1)

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

even is defined

```
Coq < Check even.
```

even

: nat -> Prop

```
Theorem two_is_even : even 2.
```

```
Proof.
```

```
    unfold even.    reflexivity.
```

```
Qed.
```

命題を返す関数を使った偶数性の定義 (2)

```
Coq < Definition even' (n:nat) : Prop :=  
    exists k, n = double k.  
even' is defined
```

```
Theorem two_is_even : even' 2.
```

```
Proof.
```

```
  exact 1. reflexivity.
```

```
Qed.
```

これらの方法では命題として使える**基本的な語彙** (原始命題) は増えていない。

偶数の集合の帰納的定義

偶数の集合 EV は、以下のふたつの条件を満たす最小の(自然数の部分)集合である。

- 0 は EV の元である
- n が EV の元ならば $S(S\ n)$ は EV の元である

- 「ふたつの条件」だけを満たす¹集合はいくらでもある
 - ▶ 例えば 1 以外の自然数の集合
- 最小性で「ゴミ」を取り除く

帰納的定義 = 規則について閉じている + 最小性

¹「規則について閉じている」ともいう

述語「偶数性」の帰納的定義

集合を述語と思うと「述語の帰納的定義」になる

自然数 n が偶数である ($ev\ n$ と書く) とは, 以下の規則から帰納的に定義される.

- $ev\ 0$ である
- 任意の自然数 n について, $ev\ n$ ならば $ev\ (S\ (S\ n))$ である

自然演繹流で書けば

新しい原子命題 $\text{ev } n$:

$$\frac{\Gamma \vdash n : \text{nat}}{\Gamma \vdash \text{ev } n : \text{Prop}} \quad (\text{EV-P})$$

- 導入規則:

$$\frac{}{\Gamma \vdash \text{ev } 0} \quad (\text{EV-I1})$$

$$\frac{\Gamma \vdash \text{ev } n}{\Gamma \vdash \text{ev } (\text{S } (\text{S } n))} \quad (\text{EV-I2})$$

Coq での帰納的述語の定義

```
Inductive ev : nat -> Prop :=  
  | ev_0 : ev 0  
  | ev_SS : forall n:nat, ev n -> ev (S (S n)).
```

- 新しい(自然数を引数とする)命題 ev の定義
 - ▶ $nat \rightarrow Prop \dots$ 自然数に関する述語
- コンストラクタ \doteq 導入規則の名前
 - ▶ あたかも定理のように使える
- コンストラクタの型 \doteq 規則の内容
 - ▶ $forall\ n:nat \dots$ 規則のパラメータ
 - ▶ $\rightarrow \dots$ 規則の前提と結論の間の水平線

例: 「4は偶数である」

```
Coq < Check ev_0.
```

```
ev_0  
  : ev 0
```

```
Coq < Check (ev_SS 0).
```

```
ev_SS 0  
  : ev 0 -> ev 2
```

```
Coq < Check (ev_SS 0 ev_0).
```

```
ev_SS 0 ev_0  
  : ev 2
```

```
Coq < Check (ev_SS 2 (ev_SS 0 ev_0)).
```

```
ev_SS 2 (ev_SS 0 ev_0)  
  : ev 4
```

例: 「4は偶数である」

Theorem four_is_even : ev 4.

Proof.

```
  apply ev_SS.  apply ev_SS.  apply ev_0.  
  (* Or: apply (ev_SS 2 (ev_SS 0 ev_0)). *)
```

Qed.

IndProp.v

- 帰納的に定義される命題
- 証明オブジェクトを証明で使う
 - ▶ 導出についての inversion
 - ▶ 導出についての帰納法
- (帰納的に定義される) 関係
- ケーススタディ: 正則表現
- (ケーススタディ: reflection の改良)

偶数性の導出と証明オブジェクトの同型性

$\text{ev } n$ が導出 (証明) できる \iff

- 導出の最後の規則は EV-I1 で $n = 0$, もしくは
- 導出の最後の規則は EV-I2 で, ある n' について $n = S(S(n'))$ かつ, $\text{ev } n'$ である.

つまり

$E : \text{ev } n$ である \iff

- $E = \text{ev_0}$ かつ $n = 0$, もしくは
- ある n' と E' について $E = \text{ev_SS } n' E'$ かつ $n = S(S(n'))$ かつ, $E' : \text{ev } n'$ である.

導出についての場合分け

Theorem `ev_minus2`: forall n,
 ev n -> ev (pred (pred n)).

Proof.

```
intros n E.
```

```
inversion E as [| n' E'].
```

```
- (* E = ev_0 *) simpl. apply ev_0.
```

```
- (* E = ev_SS n' E' *) simpl. apply E'.
```

Qed.

この証明については、`destruct` でもうまくいく。

destruct だと失敗することも

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

Proof.

```
intros n E.  
destruct E as [| n'].  
  (* The goal is still "ev n" *)
```

一般に、ゴールが複雑な (特に述語の引数が複雑な式の場合、inversionの方が賢くて、…

inversion だとうまくいく

…しかも、ありえない場合を取り除いてくれる!

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

Proof.

```
  intros n E.
```

```
  inversion E as [| n' E'].
```

```
    (* E = ev_0 なわけがない! *)
```

```
    (* E = ev_SS n' E' *)
```

```
    apply E'.
```

Qed.

任意の自然数について $ev(S(S n))$ ならば
 $ev n$

(証明) $ev(S(S n))$ の導出について場合分け

- ev_0 の場合: ありえない.
- ev_SS の場合: 導出の前提は $ev n$ のはずなので題意が示せる.

(証明終)

もうひとつの例

Theorem one_not_even : ~ ev 1.

Proof.

intros H. inversion H. Qed.

導出に対する 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

IndProp.v

- 帰納的に定義される命題
- 証明オブジェクトを証明で使う
 - ▶ 導出についての inversion
 - ▶ 導出についての帰納法
- (帰納的に定義される) 関係
- ケーススタディ: 正則表現
- (ケーススタディ: reflection の改良)

異なる偶数性の定義の同値性

Lemma `ev_even_firsttry` : forall n,
 ev n -> exists k, n = double k.

Proof.

```
intros n E. inversion E as [| n' E'].  
- (* E = ev_0 *) exists 0. reflexivity.  
- (* E = ev_SS n' E' *) simpl.
```

```
(* n' : nat
```

```
  E' : ev n'
```

```
=====
```

```
  exists k : nat, S (S n') = double k *)
```

- この時点でつまってしまう
- が、`ev n'` に注目!

もし、 $ev\ n'$ から $\exists k, n' = double\ k$ が証明できたとしたら、うまくいく:

```
assert (I : (exists k', n' = double k') ->
           (exists k, S (S n') = double k)).
{ intros [k' Hk']. rewrite Hk'.
  exists (S k'). reflexivity. }
apply I.
(* reduce the original goal to the new one *)
```

- この状況、どこかで見たような…?

偶数に関する帰納法

$P(n)$ を自然数 n の性質について述べた命題とする

偶数に関する帰納法の原理

「任意の偶数 n について $P(n)$ 」は以下と同値

- $P(0)$ かつ
 - 任意の偶数 n' について $P(n')$ ならば $P(S(S n'))$
-
- ふたつの場合分けは偶数性の定義に対応

自然演繹流で書くと

$$\frac{\Gamma \vdash \text{ev } m \quad \Gamma \vdash P[0] \quad \Gamma, n : \text{nat}, IH : P[n] \vdash P[S(Sn)]}{\Gamma \vdash P[m]} \quad (\text{EV-E})$$

帰納法を使う

Lemma ev_even : forall n,
 ev n -> exists k, n = double k.

Proof.

intros n E.

induction E as [|n' E' IH].

- (* E = ev_0 *) exists 0. reflexivity.

- (* E = ev_SS n' E' *

with IH : exists k', n' = double k' *)

destruct IH as [k' Hk'].

rewrite Hk'. exists (S k'). reflexivity.

Qed.

induction E って?

- $E : ev\ n$ ということは,
 - ▶ $E = ev_0$ かつ $n = 0$
 - ▶ $E = ev_SS\ n'\ E'$ かつ $n = S(Sn')$ かつ $E' : ev\ n'$ (つまり n' も偶数)のいずれか.
 - E は (枝分かれしない導出木で) ある種のリスト構造をしている!
- ⇒ induction E はリストに関する帰納法のようなもの!
- 「偶数性の導出に関する帰納法」ともいう

日本語だと…

定理: 自然数 n が偶数ならば, ある k について $n = \text{double } k$ である.

証明: $\text{ev } n$ の導出に関する帰納法. 最後の規則について場合分け.

- ev_0 の場合. この時 $n = 0$. $k = 0$ とすればよい.

- ev_{SS} の場合, この時ある n' について $n = S(S n')$ かつ $\text{ev } n'$ である. 帰納法の仮定より, k' について $n' = \text{double } k'$ である.

$\text{double } (S k') = S(S(\text{double } k')) = S(S n') = n$
となるので, $k = S(k')$ とすればよい. (証明終)

IndProp.v

- 帰納的に定義される命題
- 証明オブジェクトを証明で使う
 - ▶ 導出についての inversion
 - ▶ 導出についての帰納法
- (帰納的に定義される) 関係
- ケーススタディ: 正則表現
- (ケーススタディ: reflection の改良)

命題としての関係

- 一引数の命題 (一引数述語): 「もの」の性質を表す
 - ▶ even など
- 二引数の命題 (二引数述語): 「もの」と「もの」の関係を表す
 - ▶ =

関係「以下」の帰納的定義

```
Inductive le : nat -> nat -> Prop :=  
  | le_n : forall n, le n n  
  | le_S : forall n m, (le n m) -> (le n (S m))  
Notation "m <= n" := (le m n).
```

導出規則:

$$\frac{}{n \leq n} \quad (\text{LE-N})$$

$$\frac{n \leq m}{n \leq S m} \quad (\text{LE-S})$$

「以下」に関する証明

基本的には `ev` と同じ:

- ゴールにあるならコンストラクタを `apply`
- 文脈にあるなら `inversion`

```
Theorem test_le1 : 3 <= 3.
```

```
Proof. apply le_n. Qed.
```

```
Theorem test_le2 : 3 <= 6.
```

```
Proof.
```

```
  apply le_S. apply le_S.
```

```
  apply le_S. apply le_n. Qed.
```

Theorem test_le3 : (2 <= 1) -> 2 + 2 = 5.

Proof.

intros H.

inversion H. inversion H2. Qed.

「未満」の定義

Definition $lt (n m : nat) := le (S n) m$.

Notation " $m < n$ " := $(lt m n)$.

- le を使わずに直接帰納的な定義をしたら？

IndProp.v

- 帰納的に定義される命題
- 証明オブジェクトを証明で使う
 - ▶ 導出についての inversion
 - ▶ 導出についての帰納法
- (帰納的に定義される) 関係
- ケーススタディ: 正則表現
- (ケーススタディ: reflection の改良)

正則表現

文字列集合を表す記法

- 空集合
- 空文字列 ε
- 文字
- 連結
- 和
- 繰り返し

正則言語 (有限状態オートマトンが受理する言語) と対応

アルファベット集合 T 上の正則表現を表す型
reg_exp T :

```
Inductive reg_exp (T : Type) : Type :=
| EmptySet : reg_exp T
| EmptyStr : reg_exp T
| Char : T -> reg_exp T
| App : reg_exp T -> reg_exp T -> reg_exp T
| Union : reg_exp T -> reg_exp T -> reg_exp T
| Star : reg_exp T -> reg_exp T.
```

- 通常 T は有限集合
- ここではその制限は表現されていない

文字列のマッチ

$s \sim re \dots$ 「文字列 $s : list\ T$ が $re : reg_exp\ T$ にマッチする」

$$\overline{[]} = \sim \varepsilon \quad (\text{MEMPTY})$$

$$\overline{[x]} = \sim \text{Char } x \quad (\text{MCHAR})$$

$$\frac{s_1 \sim re_1 \quad s_2 \sim re_2}{s_1 ++ s_2 \sim \text{App } re_1\ re_2} \quad (\text{MAPP})$$

$$\frac{s_1 =^{\sim} re_1}{s_1 =^{\sim} \text{Union } re_1 re_2} \quad (\text{MUNIONL})$$

$$\frac{s_2 =^{\sim} re_2}{s_2 =^{\sim} \text{Union } re_1 re_2} \quad (\text{MUNIONR})$$

$$\overline{[] =^{\sim} \text{Star } re} \quad (\text{MSTAR0})$$

$$\frac{s_1 =^{\sim} re \quad s_2 =^{\sim} \text{Star } re}{s_1 ++ s_2 =^{\sim} \text{Star } re} \quad (\text{MSTARAPP})$$

帰納的命題による定義

```
Inductive exp_match T : list T -> reg_exp T -> Prop
| MEmpty : exp_match [] EmptyStr
| MChar : forall x, exp_match [x] (Char x)
| MApp : forall s1 re1 s2 re2,
    exp_match s1 re1 ->
    exp_match s2 re2 ->
    exp_match (s1 ++ s2) (App re1 re2)
...

```

```
...
| MUnionL : forall s1 re1 re2,
    exp_match s1 re1 ->
    exp_match s1 (Union re1 re2)
| MUnionR : forall re1 s2 re2,
    exp_match s2 re2 ->
    exp_match s2 (Union re1 re2)
| MStar0 : forall re, exp_match [] (Star re)
| MStarApp : forall s1 s2 re,
    exp_match s1 re ->
    exp_match s2 (Star re) ->
    exp_match (s1 ++ s2) (Star re).
```

- EmptySet についての規則はない
- Union, Star についての規則がふたつずつ

マッチの具体例

Example `reg_exp_ex1` :

```
[1] =~ Char 1.
```

Proof.

```
  apply MChar.
```

Qed.

Example `reg_exp_ex2` :

```
[1; 2] =~ App (Char 1) (Char 2).
```

Proof.

```
  apply (MApp [1] _ [2]).
```

```
  - apply MChar.
```

```
  - apply MChar.
```

Qed.

Lemma MStar1 : forall T s (re : reg_exp T) ,
s =~ re -> s =~ Star re.

Lemma empty_is_empty : forall T (s : list T),
~ (s =~ EmptySet).

Lemma MUnion' :
forall T (s : list T) (re1 re2 : reg_exp T),
s =~ re1 \ / s =~ re2 ->
s =~ Union re1 re2.

宿題： / 午前10:30 締切

- Exercise: `ev_double` (1), `inversion_practice` (1), `ev_sum` (2), `le_exercises` (3) の `le_trans`, `0_le_n`, `le_plus_1`, `leb_iff` (2)
- 解答を書き込んだ `IndProp.v` までのファイルを全て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）。