「計算と論理」
Software Foundations
その7

五十嵐 淳

cal20@fos.kuis.kyoto-u.ac.jp

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

January 5, 2021

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

命題を返す関数による偶数性の定義(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.
```

命題を返す関数による偶数性の定義(2)

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

```
Theorem two_is_even : even' 2.
Proof.
  unfold even'. exists 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 (原子命題) 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.

- `O` 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 O 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 ev n:

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

- 導入規則:

$$\frac{}{\Gamma \vdash \mathtt{ev}\ \mathtt{O}} \quad \text{(Ev-I1)}$$

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

Coq での帰納的述語の定義

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

- 新しい (自然数を引数とする) 命題 ***even*** の定義
  - `nat -> Prop` ... 自然数に関する述語
- コンストラクタ ≒ 導入規則の名前
  - あたかも公理のように使える
- コンストラクタの型情報≒ 規則の内容
  - `n:nat` ... 規則のパラメータ
  - `H:ev n` ... 規則の前提

今までの Inductive 定義との違い

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

- nat から Prop への関数の帰納的定義
- パラメータに名前がついておらず，コンストラクタは色々な引数に適用されている (even n, even 0 など)
  - list も Type から Type への関数だったが，常に list X の形で使われていた

間違った定義の仕方

```
Coq < Inductive wrong_ev (n : nat) : Prop :=
      | wrong_ev_0 : wrong_ev 0
      | wrong_ev_SS (n : nat) (H : wrong_ev n) : wrong_ev
Toplevel input, characters 0-127:
> Inductive wrong_ev (n : nat) : Prop :=
> | wrong_ev_0 : wrong_ev 0
> | wrong_ev_SS (n : nat) (H : wrong_ev n) : wrong_ev (S
Error: In environment
wrong_ev : nat -> Prop
n : nat
Unable to unify "wrong_ev 0" with "wrong_ev n".
```

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

ev n が導出 (証明) できる $\Longleftrightarrow$

- 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, ev n' である.

つまり

E : ev n である $\Longleftrightarrow$

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

導出についての場合分け (1)

```
Theorem ev_inversion :
  forall (n : nat), ev n ->
    (n = 0) \/ (exists m, n = S (S m) /\ ev m).
Proof.
  intros n Hev.  destruct Hev as [ | m Hm].
  - left. reflexivity.
  - right. exists m. split.
    + reflexivity.  + apply Hm.
Qed.
```

導出された結論 ***ev n*** から，それを導いた前提に遡る，
この種の定理を「逆転 (inversion) の定理」とも呼ぶ

destruct の働き

n (***Hev*** の型の引数) を

- 0 の場合
  - ev_0 の型の引数として出てくる
- ***S***(***S***(...)) の場合
  - ev_SS の型の引数として出てくる

に場合わけ，つまり

- ***n*** に 0 を代入したような状況
- ***n*** に ***S***(***S***(***m***)) を代入した状況 (***m*** は intro パターンに由来)

を生成

導出についての場合分け (2)

```
Theorem ev_minus2 : forall n,
  ev n -> ev (pred (pred n)).
Proof.
  intros n E.
  destruct E as [| m E'].
  - (* E = ev_0 *) simpl. apply ev_0.
  - (* E = ev_SS n' E' *) simpl. apply E'.
Qed.
```

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

- destruct の挙動:
  - 仮定の *ev* の引数 *S*(*S n*) を 0 と *S*(*S m*) に置き換えようとする
- が，ゴールに *S*(*S n*) などないので，何も置換されない

逆転の定理を使う

```
Theorem evSS_ev : forall n,
  ev (S (S n)) -> ev n.
Proof.
  intros n H. apply ev_inversion in H.
  destruct H.
  - discriminate H.
  - destruct H as [n' [Hnm Hev]].
    injection Hnm. intro Heq.
    rewrite Heq. apply Hev.
```

タクティック inversion

逆転の定理とそれにまつわる定型的な処理 (場合分け, 矛盾した場合の除去, 等式の整理) をしてくれる

```
Theorem evSS_ev' : forall n,
  ev (S (S n)) -> ev n.
Proof.
  intros n E.
  inversion E as [| n' E'].
  (* We are in the [E = ev_SS n' E'] case! *)
  apply E'.
Qed.
```

任意の自然数 n 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)

inversion による矛盾の自動除去

```
Theorem one_not_even : ~ ev 1.
Proof.
  intros H. apply ev_inversion in H.
  destruct H as [ | [m [Hm _]]].
  - discriminate H.
  - discriminate Hm.
Qed.
```

```
Theorem one_not_even' : ~ ev 1.
Proof.
  intros H. inversion H. Qed.
```

等式に対する inversion

injection, discriminate を自動でやってくれる

```
Theorem inversion_ex1 : forall (n m o : nat),
  [n; m] = [o; o] -> [n] = [m].
Proof.
  intros n m o H. inversion H. reflexivity.
Qed.

Theorem inversion_ex2 : forall (n : nat),
  S n = O -> 2 + 2 = 5.
Proof.
  intros n contra. inversion contra.
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 $\exists k, n' = \mathit{double}\ k$ 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由来

自然演繹風に書くと

ev の除去規則:

$$\frac{\Gamma \vdash \mathrm{ev}\ \ m \qquad \Gamma \vdash P[O] \qquad \Gamma, n : nat, H : evn, IH : P[n] \vdash P[S(Sn)]}{\Gamma \vdash P[m]} \text{(Ev-E)}$$

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,
  - $E = \mathtt{ev\_0}$ かつ $n = 0$
  - $E = \mathtt{ev\_SS}\ n'\ E'$ 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!

$\Longrightarrow$ `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 $n = \mathit{double}\ k$ である.

証明: 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 ev n' である．帰納法の仮定より，ある k' について $n' = \mathit{double}\ k'$ である．$\mathit{double}\ (S\ k') = S(S(\mathit{double}\ k')) = S(S\ n') = n$ 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
  - ev 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 n : le n n
  | le_S n m (H : le n m) : (le n (S m)).
Notation "m <= n" := (le m n).
```

導出規則:

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

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

「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するとしたら？

その他，二項関係の例

```
Inductive square_of : nat -> nat -> Prop :=
  | sq (n : nat) : square_of n (n * n).

Inductive next_nat : nat -> nat -> Prop :=
  | nn (n : nat) : next_nat n (S n).

Inductive next_ev : nat -> nat -> Prop :=
  | ne_1 (n : nat) (H : ev (S n))
                      : next_ev n (S n)
  | ne_2 (n : nat) (H : ev (S (S n)))
                      : next_ev n (S (S n)).
```

どういう意味かわかりますか？

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 =~ re …「文字列 s : *list T* が re : *reg_exp T* にマッチする」

$$\frac{}{[] =\sim \varepsilon} \quad (\textsc{MEmpty})$$

$$\frac{}{[x] =\sim \texttt{Char}\ x} \quad (\textsc{MChar})$$

$$\frac{s_1 =\sim re_1 \qquad s_2 =\sim re_2}{s_1 +\!\!+ s_2 =\sim \texttt{App}\ re_1\ re_2} \quad (\textsc{MApp})$$

$$\frac{s_1 \mathrel{=\tilde{}} re_1}{s_1 \mathrel{=\tilde{}} \texttt{Union}\ re_1\ re_2} \quad (\textsc{MUnionL})$$

$$\frac{s_2 \mathrel{=\tilde{}} re_2}{s_2 \mathrel{=\tilde{}} \texttt{Union}\ re_1\ re_2} \quad (\textsc{MUnionR})$$

$$\frac{}{[] \mathrel{=\tilde{}} \texttt{Star}\ re} \quad (\textsc{MStar0})$$

$$\frac{s_1 \mathrel{=\tilde{}} re \qquad s_2 \mathrel{=\tilde{}} \texttt{Star}\ re}{s_1 +\!+ s_2 \mathrel{=\tilde{}} \texttt{Star}\ re} \quad (\textsc{MStarApp})$$

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 (x : T) : exp_match [x] (Char x)
| MApp s1 re1 s2 re2
             (H1 : exp_match s1 re1)
             (H2 : exp_match s2 re2)
           : exp_match (s1 ++ s2) (App re1 re2)
...
```

```
...
| MUnionL s1 re1 re2
                (H1 : exp_match s1 re1)
              : exp_match s1 (Union re1 re2)
| MUnionR re1 s2 re2
                (H1 : exp_match s2 re2)
              : exp_match s2 (Union re1 re2)
| MStar0 re : exp_match [] (Star re)
| MStarApp s1 s2 re
                (H1 : exp_match s1 re)
                (H2 : 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.
```

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

復習: reflection

命題 P と $b = \mathit{true}$ との同値性が成立する時，b は P を反映している，という

```
Theorem eqb_eq : forall n m : nat,
  n =? m = true <-> n = m.
```

以下のような，=? を使った定理の証明に役立つ

```
Theorem filter_not_empty_In : forall n l,
  filter (fun x => n =? x) l <> [] -> In n l.
```

一般化: 述語 reflect

```
Inductive reflect (P : Prop) : bool -> Prop :=
| ReflectT (H :   P) : reflect P true
| ReflectF (H : ~ P) : reflect P false.
```

b が *P* を反映することと *reflect P b* は同値

```
Theorem iff_reflect : forall P b,
  (P <-> b = true) -> reflect P b.
Theorem reflect_iff : forall P b,
  reflect P b -> (P <-> b = true).
```

というわけで，以下のふたつの定理は同じこと．

```
Theorem eqb_eq : forall n m : nat,
  n =? m = true <-> n = m.
```

```
Theorem eqbP : forall n m : nat,
  reflect (n = m) (n =? m = true).
```

再証明

```
Theorem filter_not_empty_In : forall n l,
  filter (fun x => n =? x) l <> [] -> In n l.
Proof.
  intros n l. induction l as [| m l' IHl'].
  - (* l = [] *)
    simpl. intros H. apply H. reflexivity.
  - (* l = m :: l' *)
    simpl. destruct (eqbP n m) as [H | H].
    + (* n = m *)
      intros _. rewrite H. left. reflexivity.
    + (* n <> m *)
      intros H'. right. apply IHl'. apply H'.
Qed.
```

reflection のご利益

- 少し証明がすっきりする
- 反映可能な命題についてはできるだけ reflect を使うようにすると，積み重ねで証明がかなりすっきりすることが知られている
- Coq ライブラリ SSReflect で推奨されている証明スタイル
  - 四色問題は SSReflect で証明された
- 教科書の第二部 (本講義では扱いません) ではかなり使う

宿題： / 午前10:30 締切

- Exercise: `ev_double` (1), `inversion_practice` (1), `ev_sum` (2), `le_exercises` (3) の `le_trans`, `O_le_n`, `le_plus_l`
- 解答が記入された `IndProp.v` を origin/master に push
- レポジトリの Report.md というファイルに以下を明記:
  - 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).