

# 「計算と論理」

## Software Foundations

### その3

五十嵐 淳

`igarashi@kuis.kyoto-u.ac.jp`

京都大学

October 30, 2012

# 宿題について

Q: `mult_1_plus` について, `rewrite` を使わずに `simpl` でもできるのですが, どちらが正しいのでしょうか?

```
Theorem mult_1_plus : forall n m : nat,  
  (1 + n) * m = m + (n * m).
```

Proof.

```
  intros n m. simpl. reflexivity.
```

```
(* simpl すら不要! *)
```

Qed.

A:

- どちらも Coq にとっては正しい証明です
  - ▶ ある意味, rewrite を使わずとも解けてしまう問題設定がよろしくない
- rewrite の練習ができればそれでよし
- 「rewrite を使わずに解けたぞ!？」と気づいたことが大事

## Q: mult\_comm のヒントをくれ

```
Theorem mult_comm : forall m n : nat,  
  m * n = n * m.
```

Proof.

```
  intros m n.  induction m as [| m'].
```

まで済んだ時点でのゴール:

- $0 * n = n * 0$ 
  - ▶ 左辺は計算すれば0になる．右辺は？
- $S m' * n = n * S m'$  (IH は  $m' * n = n * m'$ )
  - ▶ 左辺は計算すると  $m' + m' * n$  になる．IH を使えば  $m' + n * m'$  までいける．右辺は？

# もう少し欲張りな人へ

- `plus_swap` ともうひとつの補題を証明すれば, (今までに証明したことを組み合わせて) 証明できる
- もうひとつの補題が `plus_swap` を使う
- 「`assert` を使え, 帰納法使うな」とあるのは, `plus_swap` でのことです. 他でも使おうと思っってはまった人, ごめんなさい.

# 本日のメニュー

Lists.v

- 自然数のペア (ふたつ組)
- 自然数リスト
- リストに関する推論
- オプション型

# 自然数のペア

引数の数が2以上のコンストラクタを使った型定義

```
Inductive natprod : Type :=  
  pair : nat    nat    natprod.
```

- コンストラクタがひとつだけの型
- pair: 自然数ふたつをとって natprod を作る
  - ▶ product ... (集合の) デカルト積

# 射影: 要素の取り出し関数

```
Definition fst (p : natprod) : nat :=  
  match p with  
  | pair x y => x  
  end.
```

```
Definition snd (p : natprod) : nat :=  
  match p with  
  | pair x y => y  
  end.
```

- `fst ... first`
- `snd ... second`



# Notation による見慣れた表記

Notation "( x , y )" := (pair x y).

```
Definition fst' (p : natprod) : nat :=  
  match p with  
  | (x,y) => x  
  end.
```

```
Definition swap_pair (p : natprod) : natprod :=  
  match p with  
  | (x,y) => (y,x)  
  end.
```

# ペアに関する簡単な性質の証明

## 定理: Surjectivity of pairing

任意のペアは，その第一射影と第二射影の組と等しい (すなわち，組を作る操作は「上への写像」になっている.)

## Coq による表現その1

Theorem surjective\_pairing' :

forall (n m : nat),

(n,m) = (fst (n,m), snd (n,m)).

Proof. reflexivity. Qed.

# より自然な表現

## その2

Theorem surjective\_pairing :

```
forall (p : natprod), p = (fst p, snd p).
```

Proof.

```
intros p. destruct p as (n,m). reflexivity.
```

Qed.

- ひとつしかないけれど場合分け
- 新しい intro パターン
  - ▶ destruct p as [n m] でも OK

# 本日のメニュー

Lists.v

- 自然数のペア (ふたつ組)
- 自然数リスト
- リストに関する推論
- オプション型

# リストとは？

「もの」(要素)を一行に並べたような集まりを表す  
データ

リストの作り方:

- 空リスト (`nil`) 全てのリストの種 (たね)
- 既存のリストの先頭へ要素を追加する (`cons`)

# 自然数リストの型定義

```
Inductive natlist : Type :=  
  | nil : natlist  
  | cons : nat -> natlist -> natlist.
```

(自然数) リストの作り方:

- 空リスト (`nil`) はリストである
- 自然数 `n` を自然数リスト `l` の先頭に追加したもの (`cons n l`) はリストである

自然数との構造の類似に注意!

# リスト表記

- `cons` の代わりにの右結合中置演算子 `n :: l`
- 要素を列挙する表記 `[n, m, ...]`
  - ▶ `.v` を直接読むとちょっと紛らわしい

以下は全て同じリストを定義している:

```
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
```

```
Definition mylist2 := 1 :: 2 :: 3 :: nil.
```

```
Definition mylist3 := [1,2,3].
```

# リスト操作関数(1): repeat

$n$  が  $\text{count}$  個並んだリスト

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | 0 => nil
  | S count' => n :: (repeat n count')
  end.
```



## リスト操作関数(2): length

リストの長さ:

```
Fixpoint length (l:natlist) : nat :=  
  match l with  
  | nil => 0  
  | h :: t => S (length t)  
end.
```

# リスト操作関数(3): app(end)

## リストの連結

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil      => l2
  | h :: t => h :: (app t l2)
  end.
```

- app l1 l2 の(右結合)中置記法: l1 ++ l2

Example test\_app1: [1,2,3] ++ [4] = [1,2,3,4].

Example test\_app2: nil ++ [4,5] = [4,5].

Example test\_app3: [1,2,3] ++ nil = [1,2,3].

## リスト操作関数(4): hd, tl

```
hd   car, tl   cdr
```

```
Fixpoint length (l:natlist) : nat :=  
  match l with  
  | nil => 0  
  | h :: t => S (length t)  
end.
```

- 引数が `nil` の場合もエラーではなく適当な値 (ここでは `nil`) を返す

# Bags via Lists

- bags a.k.a. multiset (多重集合?): 要素が現れる回数を区別する集合
- リストで表現 (実装) する

Definition bag := natlist.

- 型の別名を定義するための Definition!

# 練習問題: Bag 操作関数

- `count v s`: bag `s` 中の要素 `v` の出現回数
- `sum a b`: ふたつの bag `a`, `b` の「和集合」
- `add v s`: bag `s` に要素 `v` を追加
- `member v s`: bag `s` 中の要素 `v` の有無
  - ▶ `count` 以外は Fixpoint を使わずに定義すべし

# 本日のメニュー

## Lists.v

- 自然数のペア (ふたつ組)
- 自然数リスト
- リストに関する推論
- オプション型

# リスト vs 自然数

```
Inductive natlist : Type :=  
  | nil : natlist  
  | cons : nat -> natlist -> natlist.
```

と

```
Inductive nat : Type :=  
  | 0 : nat  
  | S : nat -> nat.
```

- 要素を無視すれば(構造だけ)見れば同じ!

## リスト vs 自然数 (2)

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil      => l2
  | cons h t => cons h (app t l2)
  end.
```

と

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | 0 => m
  | S n' => S (plus n' m)
  end.
```



# 単純化による証明

Theorem nil\_app : forall l:natlist,  
 [] ++ l = l.

Proof.

reflexivity. Qed.

自然数の足し算と同じで以下はそう簡単ではない。

Theorem app\_nil\_end : forall l:natlist,  
 l ++ [] = l.

# 場合わけによる証明

```
Theorem tl_length_pred : forall l:natlist,  
  pred (length l) = length (tail l).
```

Proof.

```
  intros l. destruct l as [| n l'].
```

```
  Case "l = nil".
```

```
    reflexivity.
```

```
  Case "l = cons n l'".
```

```
    simpl.
```

```
    reflexivity. Qed.
```

- intro パターンに注意

# リストに関する帰納法

$P(I)$  を (自然数) リスト  $I$  について述べた命題とする

## リストに関する帰納法の原理

「任意のリスト  $I$  について  $P(I)$ 」は以下と同値

- $P(\text{nil})$  かつ
- 任意の自然数  $n$ , リスト  $I'$  について  $P(I')$  ならば  $P(n::I')$

単なる場合分けと違って,  $P(n::I')$  を示すのに, ひとつ短かいリストでは  $P$  が成立していること (つまり  $P(I')$ ) を仮定してよい

- $P(I')$  を「帰納法の仮定」(induction hypothesis, IH) と呼ぶ

# 復習・比較: 数学的帰納法

$P(n)$  を自然数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  $P(S n')$

単なる場合分けと違って,  $P(S n')$  を示すのに, ひとつ小さい数では  $P$  が成立していること (つまり  $P(n')$ ) を仮定してよい

- $P(n')$  を「帰納法の仮定」(induction hypothesis, IH) と呼ぶ

## ++ の結合律

```
Theorem app_ass : forall l1 l2 l3 : natlist,  
  (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
```

Proof.

```
intros l1 l2 l3. induction l1 as [| n l1'].
```

```
Case "l1 = nil".
```

```
  reflexivity.
```

```
Case "l1 = cons n l1'".
```

```
  simpl. rewrite -> IHl1'. reflexivity.
```

Qed.

- 足し算の結合律と比較してみよう!

# informal vs formal proofs 再び

- Coq の証明はやっぱりわかりにくい
- 途中のゴールがどう変化しているか追わないといけない
- 手で書く証明には途中のゴール(今, 何を示そうとしているのか)を書きましょう

# app の結合律の証明

定理: 任意の  $I1, I2, I3$  について

$I1 ++ (I2 ++ I3) = (I1 ++ I2) ++ I3$  である

証明:  $I1$  についての帰納法 .

- $I1 = []$  とする .

$$[] ++ (I2 ++ I3) = ([] ++ I2) ++ I3$$

を示す必要があるが, これは  $++$  の定義より明らか .

- $I1 = n :: I1'$  ただし ,

$$I1' ++ (I2 ++ I3) = (I1' ++ I2) ++ I3$$

とする .

$$(n :: I1') ++ (I2 ++ I3) = ((n :: I1') ++ I2) ++ I3$$

を示す必要があるが ,  $++$  の定義より , これは

$$n :: (I1' ++ (I2 ++ I3)) = n :: ((I1' ++ I2) ++ I3)$$

と同値 . これは帰納法の仮定より明らか . (証明終)



# 例をもうひとつ

```
Theorem app_length : forall l1 l2 : natlist,  
  length (l1 ++ l2) = (length l1) + (length l2)
```

Proof.

```
  intros l1 l2. induction l1 as [| n l1'].
```

```
  Case "l1 = nil".
```

```
    reflexivity.
```

```
  Case "l1 = cons".
```

```
    simpl. rewrite -> IHl1'. reflexivity.
```

Qed.

# もう少し複雑な例

```
Fixpoint snoc (l:natlist) (v:nat) : natlist :=
  match l with
  | nil      => [v]
  | h :: t => h :: (snoc t v)
  end.
```

```
Fixpoint rev (l:natlist) : natlist :=
  match l with
  | nil      => nil
  | h :: t => snoc (rev t) h
  end.
```

```
Theorem rev_length_firsttry :  
  forall l : natlist,  
    length (rev l) = length l.
```

Proof.

```
intros l. induction l as [| n l'].
```

```
Case "l = []".
```

```
  reflexivity.
```

```
Case "l = n :: l'".
```

```
  simpl.
```

```
  rewrite <- IHl'.
```

Admitted.

## snoc に関する補題...

Theorem length\_snoc :

```
forall (n : nat) (l : natlist),  
  length (snoc l n) = S (length l).
```

Proof.

```
intros n l. induction l as [| n' l'].
```

Case "l = nil".

```
  reflexivity.
```

Case "l = cons n' l'".

```
  simpl. rewrite -> IHl'. reflexivity.
```

Qed.

## ...を使えば突破できる!

```
Theorem rev_length : forall l : natlist,  
  length (rev l) = length l.
```

Proof.

```
intros l. induction l as [| n l'].
```

```
Case "l = nil".
```

```
  reflexivity.
```

```
Case "l = cons".
```

```
  simpl. rewrite -> length_snoc.
```

```
  rewrite -> IHl'. reflexivity. Qed.
```

# 非形式証明 (バージョン 1)

雛形に沿った冗長バージョン

補題: 任意の  $n$  と  $l$  に対し

$\text{length} (\text{snoc } l \ n) = S(\text{length } l)$  である .

証明:  $l$  に関する帰納法 .

- $l = []$  とする .

$$\text{length} (\text{snoc } [] \ n) = S(\text{length } [])$$

を示す必要があるが , これは  $\text{length}$ ,  $\text{snoc}$  の定義より明らか .

- $l = n' :: l'$  ただし ,

$$\text{length (snoc } l' \text{ } n) = S(\text{length } l')$$

とする . ここで

$$\text{length (snoc (} n' :: l' \text{) } n) = S(\text{length (} n' :: l' \text{)})$$

を示す必要があるが ,  $\text{length}$ ,  $\text{snoc}$  の定義より , これは

$$S(\text{length (snoc } l' \text{ } n) = S(S(\text{length } l'))$$

の同値であり , これは帰納法の仮定より明らか . (証明終)

定理: 任意のリスト  $l$  に対し  
 $\text{length}(\text{rev } l) = \text{length } l$

証明:  $l$  についての帰納法 .

- $l = []$  とする .

$$\text{length}(\text{rev } []) = \text{length } []$$

を示す必要があるが , これは  $\text{rev}$ ,  $\text{length}$  の定義より明らか .



- $l = n :: l'$  ただし,  $\text{length} (\text{rev } l') = \text{length } l'$  とする.

$$\text{length} (\text{rev} (n :: l')) = \text{length} (n :: l')$$

を示す必要があるが,  $\text{rev}$ ,  $\text{length}$  の定義より, これは

$$\text{length} (\text{snoc} (\text{rev } l') n) = S (\text{length } l')$$

と同値. 前の補題より, これは

$$S (\text{length} (\text{rev } l')) = S (\text{length } l')$$

と同値で, これは帰納法の仮定より明らか.

# 非形式証明 (ヴァージョン 2)

わかっている人向けの短縮バージョン

定理: 任意のリスト  $l$  に対し

$$\text{length} (\text{rev } l) = \text{length } l$$

まず,

$$\text{length} (\text{snoc } l \ n) = S (\text{length } l)$$

である (これは  $l$  に関する帰納法による) ことに注意すると, この定理は  $l$  に関する帰納法で示すことができる. 特に  $l = n :: l'$  の場合で, 上の性質を帰納法の仮定と組み合わせる.

どちらがいいかは状況・読み手によるが, ひとまず本当に慣れるまでは冗長なスタイルを使ってください.

# 便利コマンド: SearchAbout

- 前に証明した定理の名前なんて覚えていられない!!
- SearchAbout foo とかすると foo に関する定理を検索してくれる!
- proofgeneral なら C-c C-a C-b (教科書が間違っている?)

# 本日のメニュー

## Lists.v

- 自然数のペア (ふたつ組)
- 自然数リスト
- リストに関する推論
- オプション型

# オプション型

「～かもしれない型」

```
Inductive natoption : Type :=  
  | Some : nat -> natoption  
  | None : natoption.
```

- Some 5
- Some 42
- None
- ⋮

# オプション型の使い道

リストの  $n$  番目の要素を返す関数 `index`

- $n$  が大きすぎる時にどうしたらいい？

```
Fixpoint index_bad (n:nat) (l:natlist) : nat :=
  match l with
  | nil => 42 (* arbitrary! *)
  | a :: l' => match beq_nat n 0 with
                | true => a
                | false => index_bad (pred n) l'
              end
  end.
```

# オプション型を使うと...

- ふつうの返り値を示す Some
- 適当な返り値がないことを示す None

```
Fixpoint index (n:nat) (l:natlist)
  : natoption :=
  match l with
  | nil => None
  | a :: l' => match beq_nat n 0 with
                | true => Some a
                | false => index (pred n) l'
              end
  end.
```

# 条件式: if-then-else

```
...  
| a :: l' => if beq_nat n 0 then Some a  
            else index (pred n) l'  
...
```

- bool だけでなく, コンストラクタがふたつの inductive type なら何でも使える!
- 一番目なら then 節
- 二番目なら else 節



# 宿題： 11/7 午前10:00 締切

- Exercise: `snd_fst_is_swap` (1), `list_funs` (2), `alternate` (3), `bag_functions` (3), `list_exercises` (3), `beq_natlist` (2)
- 解答を書き込んだ `Lists.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）．