

# 「計算と論理」

## Software Foundations

### その3

五十嵐 淳

`cal17@fos.kuis.kyoto-u.ac.jp`

京都大学

October 24, 2017

# 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
- 「辞書」のデータ表現 (省略)

# 自然数のペア

引数がふたつ (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
  - ▶ `pair 1 2` : `natprod`
  - ▶ `pair (4 + 3) 2` : `natprod`
  - ▶ `natprod` 型の式 (の値) は必ず `pair ... ...` の形をしている
- `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 (* パターンの新記法! *)  
  end.
```

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

- `fst` ... 第一射影 (first projection)
- `snd` ... 第二射影 (second projection)

# Notation による見慣れた表記の導入

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

Definition fst' (p : natprod) : nat :=  
 match p with  
 | (x,y) => x (\* パターンでも使える! \*)  
 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、組を作る操作は全射になっている.)

## 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.

# より自然な文言

## 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
  - ▶ natprod なら必ず組の形 (n,m) をしている
- 変数を複数導入するイントロパターン
  - ▶ 既に induction の用例で出てきていますが…

# 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
- 「辞書」のデータ表現 (省略)



# 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  $[n; m; \dots]$ 
  - ▶ `.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  $count$  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.
```

参考:

```
let rec repeat n count =
  if count = 0 then []
  else n :: repeat n (count - 1)
```

## 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.
```

参考:

```
let rec length l =  
  match l with  
  | [] -> 0  
  | _ :: t => length t + 1
```

# リストを消費する関数を定義するコツ

- プログラムを書く前に，入力例を沢山考えて，それぞれ出力が何になるべきかを理解する
  - ▶ 教科書であれば Example が提供されていることも
- 基本は `nil` の場合と `h :: t` の場合分け
  - ▶ リスト引数が複数ある場合，どちらで場合分けをするか悩ましいことがある  $\implies$  色々な可能性を探る
- `h :: t` の場合，`t` に対して再帰呼び出しをした結果 (の意味) を プログラムは見ないでよく考える

# 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.
```

## 参考:

```
let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | h :: t -> h :: (append t l2)
```

app 11 12 の (右結合) 中置記法: 11 ++ 12

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

```
Definition hd (default:nat) (l:natlist) : nat :=  
  match l with  
  | nil => default  
  | h :: t => h  
  end.
```

```
Definition tl (l:natlist) : natlist :=  
  match l with  
  | nil => nil  
  | h :: t => t  
  end.
```

- 引数が `nil` であってもエラーにできないので適当な値 (`default`) を返す

# 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
- 「辞書」のデータ表現 (省略)

# 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.
```

- 要素を無視して，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.

```
  intros l.  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 (tl l).
```

Proof.

```
  intros l. destruct l as [| n l'].  
  - (* l = nil *)  
    reflexivity.  
  - (* l = cons n l' *)  
    reflexivity. Qed.
```

- イントロパターンで  $l = n :: l'$  を表現している

# リストに関する帰納法

$P(l)$  を (自然数) リスト  $l$  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  $l$  について  $P(l)$ 」は以下と同値

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

単なる場合分けと違って,  $P(n::l')$  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  $P(l')$ ) を仮定してよい

- $P(l')$  … 「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_assoc : forall l1 l2 l3 : natlist,  
  (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
```

Proof.

```
intros l1 l2 l3. induction l1 as [| n l1'].  
- (* l1 = nil *)  
  reflexivity.  
- (* l1 = cons n l1' *)  
  simpl. rewrite -> IHl1'. reflexivity.
```

Qed.

- 足し算の結合律の証明と比較してみよう!

# app の結合律の日本語による証明

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.

- $l1 = n :: l1'$  ただし,

$$(l1' ++ l2) ++ l3 = l1' ++ (l2 ++ l3)$$

とする.

$$((n :: l1') ++ l2) ++ l3 = (n :: l1') ++ (l2 ++ l3)$$

を示す必要があるが,  $++$  の定義より, これは

$$n :: ((l1' ++ l2) ++ l3) = n :: (l1' ++ (l2 ++ l3))$$

と同値. これは帰納法の仮定より明らか. (証明終)

# 数学的帰納法による証明の雛形

定理: 任意の自然数  $n$  について  $P(n)$

証明:  $n$  に関する数学的帰納法による.

- $n = 0$  の場合:

……  $P(0)$  の証明 ……

- $n = S(n')$  の場合, ただし  $P(n')$  とする:

……  $P(S(n'))$  の証明 ……  
(…帰納法の仮定より…)

# リストに関する帰納法による証明の雛形

定理: 任意のリスト  $l$  について  $P(l)$

証明:  $l$  に関する帰納法による.

- $l = []$  の場合:

……  $P([])$  の証明 ……

- $l = n :: l'$  の場合, ただし  $P(l')$  とする:

……  $P(n :: l')$  の証明 ……

(…帰納法の仮定より…)

# もう少し複雑な例: リストの反転

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

- 後ろに要素を追加するのに `append` を使っている

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

Proof.

```
intros l. induction l as [| n l'].  
- (* l = [] *)  
  reflexivity.  
- (* l = n :: l' *)  
  simpl.
```

(\* 無理っぽいゴール:

```
length (rev l' ++ n) = S (length l') *)
```

- 我々は `append` と `length` の関係に関して何も示していない!

## こういう補題を立てれば…

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

Proof.

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

```
- (* l1 = nil *)
```

```
  reflexivity.
```

```
- (* l1 = cons n l1' *)
```

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

Qed.

つまったゴールより少し一般化(?)した定理になっている



## …突破できる!

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

Proof.

```
intros l. induction l as [| n l'].  
- (* l = nil *)  
  reflexivity.  
- (* l = cons *)  
  simpl. rewrite -> app_length, plus_comm.  
  simpl. rewrite -> IHl'. reflexivity. Qed.
```

- `rewrite` で使う定理が複数指定できる
  - ▶ 左から順に適用
  - ▶ 方向もひとつひとつ指定できる

# 非形式証明 (ヴァージョン 1)

「雛形」に沿った冗長バージョン

補題: 任意の  $I1, I2$  に対し

$length (I1 ++ I2) = length I1 + length I2$   
である.

証明:  $I1$  に関する帰納法. (以下  $length$  は  $len$  と略す.)

- $I1 = []$  とする.

$$len ([] ++ I2) = length [] + length I2$$

を示す必要があるが, これは  $len, +$  の定義より明らか.

- $l1 = n :: l1'$  ただし,

$$\text{length } (l1' ++ l2) = \text{len } l1' + \text{len } l2$$

とする. ここで

$$\text{len } ((n :: l1') ++ l2) = \text{len } (n :: l1') + \text{len } l2$$

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

$$S(\text{len}(l1' ++ l2)) = S(\text{len } l1' + \text{len } l2)$$

と同値であり, 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  
 $length (rev l) = length l$

証明:  $l$  についての帰納法.

- $l = []$  とする.

$$length (rev []) = length []$$

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

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

$$length (rev (n :: l')) = length (n :: l')$$

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

$$length ((rev l') ++ [n]) = S (length l')$$

と同値. 前の補題より, これは

$$length (rev l') + 1 = S (length l')$$

と同値で, これは  $+$  の交換律, 帰納法の仮定などより明らか.

# 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,  $\text{length}(l ++ [n]) = S(\text{length } l)$  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: Search

- 前に証明した定理の名前なんて覚えていられない!
- Search 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-a で検索, 検索結果は C-c C-; でペーストできる.

# 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
- 「辞書」のデータ表現 (省略)



# 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 `nth`

- $n$  が大きすぎる時にどうしたらいい？

```
Fixpoint nth_bad (n:nat) (l:natlist) : nat :=
  match l with
  | nil => 42 (* arbitrary! *)
  | a :: l' => match beq_nat n 0 with
                | true => a
                | false => nth_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 nth_error (l:natlist) (n:nat)
  : natoption :=
match l with
| nil => None
| a :: l' => match beq_nat n 0 with
              | true => Some a
              | false => nth_error l' (pred n)
            end
end.
```

# 条件式: if-then-else

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

- 実は `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!
  - ▶ 定義での順番依存
    - ★ 一番目のコンストラクタなら `then` 節, 二番目なら `else` 節
- パターンによる値の取り出しはできない

# 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
- 「部分写像」のデータ表現 (省略)
  - ▶ partial maps, dictionary, または連想リスト (association list) のデータ構造に関する定義と練習問題

# 宿題： 11/ 午前 10:30 締切

- Exercise: `snd_fst_is_swap` (1), `list_funs` (2), `list_exercises` (3), `beq_natlist` (2), `hd_error` (2)
- その他は随意課題
- 解答を書き込んだ `Basics.v`, `Induction.v`, `Lists.v` を含む zip ファイルを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

# 宿題のヒント

- `list_exercises` の `rev_involutive` はまあまあ難しい!
- 補題をうまく設定するのがコツ (ゴールの意味をよく考えて!)
  - ▶ `rev_involutive` では `rev` と `append` の関係について