

# 「計算と論理」

## Software Foundations

### その4

五十嵐 淳

`cal15@fos.kuis.kyoto-u.ac.jp`

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

京都大学

November 5, 2015

# Poly.v

- 多相性
  - ▶ 多相的リスト
    - ★ 多相的型定義と多相的関数定義
    - ★ 型宣言の推論と型引数の推論
    - ★ 型引数の暗黙化指定
    - ★ 多相的リストに関する証明
  - ▶ 多相的ペア
  - ▶ 多相的オプション型
- データとしての関数

# 問題: 真偽値リスト型を定義せよ

自然数リストをマスターした今なら朝飯前(ですね?)

```
Inductive boollist : Type :=  
  | bool_nil : boollist  
  | bool_cons : bool -> boollist -> boollist.
```

さて, boollist 用の関数定義と証明を...

- って, natlist の時と同じことの繰り返しでは!?
- 違いは要素型, コンストラクタ, 型の名前だけ

⇒ 型定義の共通化

- ▶ 型  $\mapsto$  その型を要素とするリスト型定義

# 型パラメータによる型定義の抽象化

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

- リストの要素型を表す型パラメータ  $X$
- $\text{list } T$  は型  $T$  の値を要素とするリストの型
  - ▶  $\text{list nat}$  は  $\text{nat}$  を要素とするリスト型
  - ▶  $\text{list bool}$  は  $\text{bool}$  を要素とするリスト型
- “list” は単体では型ではないことに注意!

# リストの作り方

要素型をコンストラクタに型引数として与える

```
Coq < Check (nil nat).
```

```
nil nat
```

```
  : list nat
```

```
Coq < Check (cons nat 1 (nil nat)).
```

```
cons nat 1 (nil nat)
```

```
  : list nat
```

```
Coq < Check (cons bool true (cons bool false (nil bool))).
```

```
cons bool true (cons bool false (nil bool))
```

```
  : list bool
```

# 型引数によって型が変わる `nil/cons`

```
Coq < Check (cons nat).  
cons nat  
      : nat -> list nat -> list nat
```

```
Coq < Check (cons bool).  
cons bool  
      : bool -> list bool -> list bool
```

では, `cons` の単体での型は?

# 多相的コンストラクタと型の全称量化

```
Coq < Check cons.
```

```
cons
```

```
  : forall X : Type, X -> list X -> list X
```

cons は「任意の型  $X$  について

$X \rightarrow \text{list } X \rightarrow \text{list } X$ 」

という型の (二引数) 多相的コンストラクタ

- 多相的 (polymorphic) … 「(型が) 色々変わる」の意
- $X$  に具体的な型を与えると、それに応じて型が変化  
する

# 多相的型定義

`list X` のような，型パラメータで抽象化された型定義を多相的型定義と呼ぶ

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

# 個別のリスト処理関数から…

```
Fixpoint length_nats (l:natlist) : nat :=
  match l with
  | nat_nil      => 0
  | nat_cons h t => S (length_nats t)
  end.
```

```
Fixpoint length_bools (l:boollist) : nat :=
  match l with
  | bool_nil      => 0
  | bool_cons h t => S (length_bools t)
  end.
```

(natlist のコンストラクタの名前は変えました)

## …多相的リスト処理関数へ

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

- 型定義と同様に関数定義も要素型で抽象化
- 型  $X$  と, その型を要素とするリスト  $l$  が引数
- $\text{length } T \cdots \text{list } T$  型のリストの長さを計算 (扱うリストの要素型が固定される)

# length の使用例

多相コンストラクタと同様，型引数を与える

Example test\_length1 :

```
length nat
```

```
(cons nat 1 (cons nat 2 (nil nat))) = 2.
```

Proof. reflexivity. Qed.

Example test\_length2 :

```
length bool (cons bool true (nil bool)) = 1.
```

Proof. reflexivity. Qed.

# length の型

```
Coq < Check (length nat).
```

```
length nat  
  : list nat -> nat
```

```
Coq < Check (length bool).
```

```
length bool  
  : list bool -> nat
```

```
Coq < Check length.
```

```
length  
  : forall X : Type, list X -> nat
```

任意の要素のリストを受け取れることを示している

# その他の関数定義例

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

- パターン `cons h t` では要素型に言及する必要はない (`l1` の型から自明なので)
- (`snoc`, `rev` も同様)

# Poly.v

- 多相性
  - ▶ 多相的リスト
    - ★ 多相的型定義と多相的関数定義
    - ★ 型宣言の推論と型引数の推論
    - ★ 型引数の暗黙化指定
    - ★ 多相的リストに関する証明
  - ▶ 多相的ペア
  - ▶ 多相的オプション型
- データとしての関数

# 型宣言の推論

パラメータや関数の返り値型の宣言を省略した時に適当な型をみつくりたい機能

```
Coq < Fixpoint app' X l1 l2 := (* ": ..." がない *)
Coq <   match l1 with
Coq <   | nil          => l2
Coq <   | cons h t    => cons X h (app' X t l2)
Coq <   end.
```

省略前の `app` と同じ型!

```
Coq < Check app'.
app'
  : forall X : Type, list X -> list X -> list X
```

# どれくらい宣言すればいいの？

- 宣言の意義: 書き手の意図のシステム・読み手への伝達
  - 多すぎるのもかえってわずらわしい
  - 少なすぎると読み手の負担が増える
- ⇒ バランスが大事・自分のスタイルを見つけましょう
- この教科書では, トップレベル関数のパラメータの型宣言は書き下す.

# 型引数の推論

多相関数に渡す型引数の指定

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

```
Check (cons nat 1 (nil nat)).
```

- この青い部分をいちいち書くのは面倒!
- Coq に推論させよう!

```
Fixpoint length' (X:Type) (l:list X) : nat :=
  match l with
  | nil      => 0
  | cons h t => S (length' _ t)
  end.
```

Check (cons \_ 2 (cons \_ 1 (nil \_))).

- \_ (アンダースコア) … 「ここに適当に何か入れてください」

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

は

```
Fixpoint length (X:_) (l:_) : _ :=
  match l with
  | nil          => 0
  | cons h t    => S (length X t)
  end.
```

と同じこと.

# 「アンダースコア打つの面倒…」

引数の暗黙化 or どこでもアンダースコアを省略するためのおまじない

```
Arguments nil {X}.
```

```
Arguments cons {X} _ _.
```

```
Arguments length {X} 1.
```

- 以後、パラメータ  $x$  は**必ず**省略するよ、という宣言
- `nil` は `nil _` の、`cons` は `cons _` のことになる

```
Definition list123'' :=  
  cons 1 (cons 2 (cons 3 nil)).
```

```
Check (length list123''').
```

- 多相性
  - ▶ 多相的リスト
    - ★ 多相的型定義と多相的関数定義
    - ★ 型宣言の推論と型引数の推論
    - ★ 型引数の暗黙化指定
    - ★ 多相的リストに関する証明
  - ▶ 多相的ペア
  - ▶ 多相的オプション型
- データとしての関数

# 関数引数の暗黙化

引数宣言を  $()$  ではなく  $\{\}$  で囲むと暗黙の引数になる:

```
Fixpoint length'' {X:Type} (l:list X) : nat :=
  match l with
  | nil          => 0
  | cons h t    => S (length'' t)
  end.
```

- 再帰呼び出しで型引数が省略されている
- 教科書では Arguments ではなく, なるべくこちらを使う

# 注意: 型定義の型パラメータの暗黙化

```
Inductive list' {X:Type} : Type :=  
  | nil' : list'  
  | cons' : X -> list' -> list'.
```

とすると, `list' nat` とか書けなくなるので NG.  
⇒ 暗黙化は型定義で行わず, コンストラクタだけ別途  
Arguments で暗黙化する.

# 推論は失敗することもある

Definition mynil := nil. (\* nil \_ のこと! \*)

- nil への型引数を推論しようとするが、決め手がないのでエラー

⇒ 回避策

- ▶ ヒントを与える

```
Definition mynil : list nat := nil.
```

- ▶ 暗黙化を無効化するオペレータ @ を使う

```
Definition mynil' := @nil nat.
```

# 短縮表記の定義

多相的リストの場合，暗黙の引数があってはじめて可能!

Notation `"x :: y" := (cons x y)`  
(at level 60, right associativity).

Notation `"[ ]" := nil.`

Notation `"[ x ; .. ; y ]" :=`  
(cons x .. (cons y []) ..).

Notation `"x ++ y" := (app x y)`  
(at level 60, right associativity).

Definition `list123''' := [1; 2; 3].`

- 多相性
  - ▶ 多相的リスト
    - ★ 多相的型定義と多相的関数定義
    - ★ 型宣言の推論と型引数の推論
    - ★ 型引数の暗黙化指定
    - ★ 多相的リストに関する証明
  - ▶ 多相的ペア
  - ▶ 多相的オプション型
- データとしての関数

# 多相リストに関する性質

要素型毎に証明してもよいけど…

```
Theorem nil_app_nat :
```

```
  forall l:list nat, [] ++ l = l.
```

```
Proof. intros l. reflexivity. Qed.
```

```
Theorem nil_app_bool :
```

```
  forall l:list bool, [] ++ l = l.
```

```
Proof. intros l. reflexivity. Qed.
```

…型に関する全称量化を使うとまとめて証明できる!

```
Theorem nil_app :
```

```
  forall X:Type, forall l:list X,  
    [] ++ l = l.
```

```
Proof.
```

```
  intros X l. reflexivity.
```

```
Qed.
```

- データについての「任意の～」と同様に `intros` を使い，型 `X` を仮定する
- 型全体の集合 (`Type` の全貌) が何かわかっていないのでやや気持ち悪い？

# Poly.v

- 多相性
  - ▶ 多相的リスト
  - ▶ 多相的ペア
  - ▶ 多相的オプション型
- データとしての関数

# 多相的ペア

第一要素, 第二要素それぞれの型を表す, ふたつの型  
パラメータ

```
Inductive prod (X Y : Type) : Type :=  
  pair : X -> Y -> prod X Y.
```

```
Arguments pair {X} {Y} _ _.
```

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

```
Notation "X * Y" := (prod X Y) : type_scope.
```

- 型表記の短縮形定義  $X * Y$  (かけ算ではない!)
- $( , )$  と  $*$  の違いに注意
  - ▶  $(1, \text{true}) : \text{nat} * \text{bool}$

# 多相的射影関数

```
Definition fst {X Y : Type} (p : X * Y) : X :=  
  match p with (x,y) => x end.
```

# 多相オプション型

```
Inductive option (X:Type) : Type :=  
  | Some : X -> option X  
  | None : option X.
```

```
Arguments Some {X} _.
```

```
Arguments None {X}.
```

# 多相的 index 関数

```
Fixpoint index {X : Type} (n : nat)
  (l : list X) : option X :=
  match l with
  | [] => None
  | a :: l' => if beq_nat n 0 then Some a
               else index (pred n) l'
  end.
```

Example test\_index1 :

index 0 [4;5;6;7] = Some 4.

Example test\_index3 : index 2 [true] = None.

# 今日のメニュー

## Poly.v

- 多相性
- データとしての関数
  - ▶ 高階関数
  - ▶ 部分適用
  - ▶ より道: カリー化 (省略)
  - ▶ 高階関数カタログ
    - ★ フィルター
    - ★ 匿名関数
    - ★ マップ
    - ★ 畳み込み (fold)
  - ▶ 関数を作る関数

# 高階関数

- Coq では、他の関数型言語 (Scheme, OCaml, Haskell, ...) と同様、関数は「ふつうの」データとして
  - ▶ 引数として他の関数に渡したり
  - ▶ 関数の返り値として返したり
  - ▶ データ構造に入れたり、というように使える
- 一階の関数: (関数でない) データからデータへの関数
- 二階の関数: 一階の関数を引数とする関数
- 三階の関数: 二階の関数を引数とする関数
- ⋮

## 例:

自然数上の関数  $f$  と  $x$  を受け取って、 $f$  を三回適用した結果を返す関数

```
Definition doit3times_nat (f:nat->nat)
                          (n:nat) : nat :=
  f (f (f n)).
```

```
Example test_doit3times_nat:
  doit3times_nat minustwo 9 = 3.
Proof. reflexivity. Qed.
```

# 多相バージョン

`nat` に限らず，引数と返り値の型が同じ関数なら同じことができる

```
Definition doit3times {X:Type}
  (f:X->X) (n:X) : X :=
  f (f (f n)).
```

```
Example test_doit3times':
  doit3times negb true = false.
Proof. reflexivity. Qed.
```

# 部分適用

二引数関数の型  $\text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$  の本当の読み方:

$$\text{nat} \rightarrow (\text{nat} \rightarrow \text{nat})$$

- $\rightarrow$  は (ペア型の  $*$  のような) **型上の二項演算子** (右結合)
- $\text{nat}$  をひとつ受け取ると  $\text{nat} \rightarrow \text{nat}$  の値 ( $\leftarrow$ 関数!) を返す関数
- 引数はふたつ同時に与えなくてもよい!

$\implies$  部分適用

# 例

Definition `plus3 := plus 3.`

Check `plus3.`

Example `test_plus3 : plus3 4 = 7.`

(\* `plus 3 4` は `(plus 3) 4` のこと  
つまり、関数適用は左結合 \*)

Example `test_plus3' : doit3times plus3 0 = 9.`

Example `test_plus3'' :`

`doit3times (plus 3) 0 = 9.`

# 今日のメニュー

## Poly.v

- 多相性
- データとしての関数
  - ▶ 高階関数
  - ▶ 部分適用
  - ▶ より道: カリー化 (省略)
  - ▶ 高階関数カタログ
    - ★ フィルター
    - ★ 匿名関数
    - ★ マップ
    - ★ 畳み込み (fold)
  - ▶ 関数を作る関数

# 高階関数カタログ(1): filter

リスト l 中の test を満たす要素のみを残す

```
Fixpoint filter {X:Type}
  (test: X->bool) (l:list X) : (list X) :=
  match l with
  | []      => []
  | h :: t =>
    if test h then h :: (filter test t)
    else          filter test t
  end.
```

Example test\_filter1:

```
filter evenb [1;2;3;4] = [2;4].
```

# 匿名関数

Scheme でいうと lambda のこと

- Scheme: (lambda (x) 式)
- Coq: fun x => 式

以上.

# 匿名関数(もう少し丁寧に)

- 高階関数に渡す関数には, わざわざ定義をする(名前をつける)価値がないものもしばしば
- 使う(渡す)ところで “on the fly” で作る, 名前のない匿名関数

Example `test_filter2'`:

```
filter (fun l => beq_nat (length l) 1)
      [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
= [ [3]; [4]; [8] ].
```

# 複数引数の匿名関数

```
Coq < Check (fun x y => x + y + 1).
```

```
fun x y : nat => x + y + 1  
  : nat -> nat -> nat
```

```
Coq < Check (fun (x y : nat) => x + y + 1).
```

```
fun x y : nat => x + y + 1  
  : nat -> nat -> nat
```

```
Coq < Check (fun (b : bool) (x : nat) =>
```

```
  Coq < if b then x else x + 1).
```

```
fun (b : bool) (x : nat) => if b then x else x + 1  
  : bool -> nat -> nat
```

## 高階関数カタログ(2): map

リスト  $l = [x_1; \dots; x_n]$  の各要素に関数  $f$  を適用したもものからなるリスト  $[f\ x_1; \dots; f\ x_n]$  を返す

```
Fixpoint map {X Y:Type} (f:X->Y) (l:list X)
  : (list Y) :=
  match l with
  | []      => []
  | h :: t => (f h) :: (map f t)
  end.
```

# 例

Example test\_map1:

```
map (plus 3) [2;0;2] = [5;3;5].
```

Example test\_map2:

```
map oddb [2;1;2;5] = [false;true;false;true].
```

- 関数  $f$  の引数・返り値の型は違っていてもよい
  - ▶ それぞれ  $X, Y$  に相当

# 高階関数カタログ(3): fold

```
Fixpoint fold {X Y:Type}
  (f: X->Y->Y) (l:list X) (b:Y) : Y :=
  match l with
  | nil => b
  | h :: t => f h (fold f t b)
  end.
```

与えられた  $l$  の

- $nil$  は  $b$  で
- $cons$  は  $f$  で置き換える
  - ▶  $| cons\ h\ t\ =>\ f\ h\ (fold\ f\ t\ b)$  の方が置き換えている感じがでる？

# 例

```
Example fold_example0 :  
  fold plus (1 :: 2 :: 3 :: 4 :: nil) 0  
    = 1 + (2 + (3 + (4 + 0))).
```

```
Example fold_example1 :  
  fold mult [1;2;3;4] 1 = 24.
```

```
Example fold_example2 :  
  fold andb [true;true;false;true] true = false
```

```
Example fold_example3 :  
  fold app [[1];[];[2;3];[4]] [] = [1;2;3;4].  
(* [1] ++ [] ++ [2;3] ++ [4] ++ [] *)
```

# 今日のメニュー

Poly.v

- 多相性
- データとしての関数
  - ▶ 高階関数
  - ▶ 部分適用
  - ▶ より道: カリー化 (省略)
  - ▶ 高階関数カタログ
  - ▶ 関数を作る関数

# 関数を作る関数

- 部分適用で見たように、二引数関数は「一関数を返す関数」と考えられる
- 関数を返す関数のもっと積極的な(?)使い方を見てみよう
  - ▶ 定数関数を作る関数
  - ▶ 関数の挙動を一部変更する関数

# 定数関数を作る関数

```
Definition constfun {X: Type}
  (x: X) : nat->X :=
  fun (k:nat) => x.
```

```
Definition constfun' (* これでも同じ *)
  {X: Type} (x: X) (k: nat) : X := x.
```

```
Definition ftrue := constfun true.
(* a function that always returns true *)
```

```
Example constfun_example1: ftrue 0 = true.
```

```
Example constfun_example2: (constfun 5) 99 = 5.
```

# 関数の一部の戻り値の変更

- $f$ : 自然数を定義域とする関数
- 自然数  $k$  と何らかのデータ  $x$  に対し, 新しい関数  $f[k \mapsto x]$  を以下のように定義:

$$f[k \mapsto x] \stackrel{\text{def}}{=} g \text{ s.t. } \begin{cases} g(k) = x \\ g(n) = f(n) \text{ (if } n \neq k) \end{cases}$$

Definition override  $\{X: \text{Type}\}$

```
(f: nat->X) (k:nat) (x:X) : nat->X :=  
fun (k':nat) => if beq_nat k k' then x  
                else f k'.
```

```
Definition fmostlytrue :=  
  override (override ftrue 1 false) 3 false.
```

```
Example override_example1 :  
  fmostlytrue 0 = true.
```

```
Example override_example2 :  
  fmostlytrue 1 = false.
```

```
Example override_example3 :  
  fmostlytrue 2 = true.
```

```
Example override_example4 :  
  fmostlytrue 3 = false.
```

- この後教科書では `override` を沢山使います
- 色々性質を証明します
- そのためにもう少しタクティックを覚えましょう (次章)
- (…といっても、この講義でカバーする教科書の範囲では、もう `override` は出てこないのですが…)

# unfold タクティック

(Definition による) 定義を展開するタクティック

```
Theorem unfold_example : forall m n,  
  3 + n = m ->  
  plus3 n + 1 = m + 1.
```

Proof.

```
intros m n H.
```

(\* 3 + n と plus3 n は定義

Definition plus3 x := (plus 3) x.

により等しいが、(見た目)は違う \*)

```
unfold plus3.
```

```
rewrite -> H.
```

```
reflexivity. Qed.
```

```
Theorem override_eq :  
  forall {X:Type} x k (f:nat->X),  
    (override f k x) k = x.
```

Proof.

```
intros X x k f.
```

(\* simpl. では Definition は展開されない! \*)

(\* compute. なら展開するが,  
 ゴールはもはや理解不能 \*)

```
unfold override.
```

```
rewrite <- beq_nat_refl.
```

```
reflexivity.
```

Qed.

# 宿題 : 1?/??(火) 午前 10:30 締切

- Exercise: `poly_exercises` (2), `split` (2), `flat_map` (2) `override_neq` (2)
- 解答を書き込んだ `Basics.v`, `Induction.v`, `Lists.v`, `Poly.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 (URL など).