「計算と論理」
Software Foundations
その 4

五十嵐 淳
igarashi@kuis.kyoto-u.ac.jp

京都大学

November 6, 2012

コメント欄より

assert の使い方がのみこめません

- 補助定理を立てたい機会はまだ少ないので，無理して使わなくてもよい
- 補助定理が仮定されたものに言及しない場合はなおさら
- 既に証明された一般的な性質の具体例を仮定に追加したい時にわりとよく使う
  - 教科書の例 (plus_rearrange) はその一例

コメント欄より

`rewrite` の方向指定が当てずっぽうになりがちです

- 矢印は `<-` が使う等式の右から左，`->` が左から右への書き換え
- どっちが左辺かわからなくなったら，`Check` を使って確かめましょう．
  - 証明中でも `Check` は使える
  - ProofGeneral なら `C-c C-v` でコマンドが入力できる

assert の応用(再掲)

```
Theorem plus_rearrange : forall n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  assert (H: n + m = m + n).
    (* n と m の交換に特化 *)
    Case "Proof of assertion".
    rewrite -> plus_comm. reflexivity.
  rewrite -> H. reflexivity. Qed.
```

前回のメニュー

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.
```

natoption **型の値**:

- None
- Some 5
- Some 42
- ⋮

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 O 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 O 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

今日のメニュー

Poly.v 前半

- 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

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!?
- 違いは要素の型とコンストラクタ・型の名前だけ

$\Longrightarrow$ 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

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

- リストの要素型を表す型パラメータ `X`
- `list X`: `X` を要素とするリストの型
  - `list nat` は `nat` を要素とするリスト型
  - `list bool` は `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

リストの要素型をコンストラクタに与える

```
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
```

nat, bool は**型引数**と呼ばれる

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 -> list X -> 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**
- **具体的な X の「値」を与えると，それに応じて型が変化**

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.
```

多相的リスト処理関数

```
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
- `length T` で `list T` 型のリストの長さを計算

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
```

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.
```

- **パターンでは要素型に言及する必要はない (自明なので)**

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ろう機能

```
Coq < Fixpoint app' X l1 l2 :=
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

$\Longrightarrow$ バランスが大事・自分のスタイルを見つけましょう

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       => O
  | cons h t => S (length X t)
  end.
```

は

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

と同じこと.

暗黙の引数

どこでもアンダースコアを省略するためのおまじない

```
Implicit Arguments nil [[X]].
Implicit Arguments cons [[X]].
Implicit Arguments length [[X]].

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

- 以後，パラメータ `X` は (必ず) 省略するよ，という宣言
- `nil` は `nil _` のことになる

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 Implicit Arguments ではなく，なるべくこちらを使う**
  - **例外: コンストラクタ型引数の暗黙化**

推論は失敗することもある

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

- 型引数を推論しようとするが，決め手がないのでエラー

$\Longrightarrow$ 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
- 型全体が何かわかっていないのでやや気持ち悪い？

今日のメニュー

Poly.v **前半**

- **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.

Implicit 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, true) :  nat * bool`

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.

Implicit Arguments Some [[X]].
Implicit 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 O then Some a
                  else index (pred n) l'
  end.

Example test_index1 :
  index O [4,5,6,7]  = Some 4.
Example test_index3 : index 2 [true]  = None.
```

今日のメニュー

Poly.v **前半**

- **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 (来週にまわす)**

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 `nat -> nat -> nat` **の本当の読み方**:

```
nat -> (nat -> nat)
```

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

$\Longrightarrow$ **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 **前半**

- 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 (**来週にまわす**)

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 = [x1, ..., xn]` **の各要素に関数** `f` **を適用したものからなるリスト** `[f x1, ..., f xn]` **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] ++ [] *)
```

宿題： 11/14 午前10:00 締切

- Exercise: `split` (2), `hd_opt_poly` (1), `filter_even_gt7` (2), `partition` (3), `flat_map` (2)
- **解答を書き込んだ `Poly.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)．**