「計算と論理」
Software Foundations
その 4

五十嵐 淳

cal18@fos.kuis.kyoto-u.ac.jp

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

京都大学

November 6, 2018

課題1の質問など(やっと)見ました!

`andb_true_elim2` で場合分けをすると前提が偽になるケースがある:

- Q1: なんとか証明したが変な感じがする
- Q2: 前提が偽であるならすぐに証明が終了できるべきでは？

- A1: $c = \mathit{false}$ の場合は，実質的に「$\mathit{false} = \mathit{true}$ ならば $\mathit{false} = \mathit{true}$」を証明するわけですが，これは「$a = b$ ならば $a = b$」の形をしているので，自明な命題を rewrite を使って証明しただけとも考えられます.
- A2: それはその通り．そのうちそういうタクティックも教えます．(自分で調べた人はそれで OK．)

- Q: destruct を使ってた時は型の定義のしかたに基づいて場合わけのパターンが決まるというようなことを言っていたと思うが、自然数を例えば偶数 or 奇数みたいな場合分けにしたい時はどうするんだろう？専用の書き方があるのかな？などということを思ったりしました。
- A: 偶数/奇数概念を適切に定義すればできます．質問とはちょっと違いますが,「任意の偶数について〜」のような命題の証明については，後で時間を割いてやることになります.

資料の訂正

その 1:

誤) rewrite は何箇所も同時に書き換える場合がある

正) rewrite は適用可能箇所が複数あっても一箇所だけ書き換える

言い訳: 昔のバージョンは同時に書き換えていた記憶が…

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
  | bool_cons (b : bool) (l : 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

$\Longrightarrow$ 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

```
Inductive list (X:Type) : Type :=
  | nil
  | cons (x : X) (l : list X).
```

- リストの要素型を表す型パラメータ X
- list T は型 T の値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

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 -> 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」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
  | cons (x : X) (l : list X).
```

個別のリスト処理関数から…

```
Fixpoint repeat_nat (x : nat) (count : nat) : natlist :=
  match count with
  | O => nat_nil
  | S count' => nat_cons x (repeat_nat x count')
  end.
Fixpoint repeat_bool (x : bool) (count : nat)
                                        : boollist :=
  match count with
  | O => bool_nil
  | S count' => bool_cons x (repeat_bool x count')
  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 repeat (X : Type) (x : X)
                (count : nat) : list X :=
  match count with
  | O => nil X
  | S count' => cons X x (repeat X x count')
  end.
```

- 型定義と同様に関数定義も要素型で抽象化
- 関数・コンストラクタを使うところには要素型情報を供給

repeat 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_repeat1 :
  repeat nat 4 2
  = cons nat 4 (cons nat 4 (nil nat)).
Proof. reflexivity.  Qed.

Example test_repeat2 :
  repeat bool false 1
  = cons bool false (nil bool).
Proof. reflexivity.  Qed.
```

repeat の型

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

Coq < Check (repeat bool).
repeat bool
     : bool -> nat -> list bool

Coq < Check repeat.
repeat
     : forall X : Type, X -> nat -> list X
```

任意の要素から，そのリストを作れることを示している

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ろう機能

```
Coq < Fixpoint repeat' X x count : list X :=
        match count with
        | O        => nil X
        | S count' => cons X x (repeat' X x count')
        end.
```

省略前の repeat と同じ型!

```
Coq < Check repeat'.
repeat'
     : forall X : Type, X -> nat -> 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

- この教科書では，(`Definition`, `Fixpoint` で定義する) 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 repeat' X x count : list X :=
  match count with
  | O          => nil X
  | S count' => cons X x (repeat' X x count')
  end.

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

- この青い部分をいちいち書くのも面倒!
- Coq に推論させよう!

```
Fixpoint repeat'' X x count : list X :=
  match count with
  | O          => nil X
  | S count' => cons _ x (repeat'' _ x count')
  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」

「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 repeat {X} x count.
```

- 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 repeat''' {X : Type} (x : X)
                        (count : nat) : list X :=
  match count with
  | O          => nil
  | S count' => cons x (repeat''' x count')
  end.
```

- 再帰呼び出しで型引数が省略されている (省略しないといけない)
- 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'
  | cons' (x : X) (l : 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.

その他の多相的関数

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

Fixpoint rev {X:Type} (l:list X) : list X :=
  match l with
  | nil       => nil
  | cons h t => app (rev t) (cons h nil)
  end.
```

```
Fixpoint length {X : Type} (l : list X)
                                    : nat :=
  match l with
  | nil => 0
  | cons _ l' => S (length l')
  end.
```

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

$\Longrightarrow$ 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.
  ```

(やっと)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？
  - X についての場合わけとかはできません

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 : X) (y : 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, true) : nat * 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 : X)
  | None.

Arguments Some {X} _.
Arguments None {X}.
```

多相的 nth 関数

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

Example test_index1 :
  nth_error 0 [4;5;6;7]  = Some 4.
Example test_index3 :
  nth_error 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
  - 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 (OCaml, Haskell, Scheme, ...) 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 n 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)      (x:X)    : X    :=
  f (f (f x)).

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

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
  - 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

OCaml でいうと `fun` のこと

- OCaml: `fun 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 => (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) =>
               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 (fun x => plus 3 x) [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
  | [] => 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
  - 高階関数カタログ
  - 関数を作る関数

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.
```

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 (partial application)

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.
```

宿題：11/　(火) 午前10:30 締切

- Exercise: `poly_exercises` (2), `split` (2), `flat_map` (2) `currying` (2)
- 解答が記入された `Poly.v` を origin/master に push
- レポジトリに「課題4」という名前の新規 issue を作成し以下を明記:
  - 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).