软件理论基础与实践

Poly: Polymorphism and Higher-Order Functions

熊英飞
北京大学

多态(Polymorphism)

- 历史上，由于函数式程序设计语言和面向对象程序设计语言独立发展，多态一词被长期混用
- 至少有三种不同的多态
- Ad-hoc Polymorphism
  - 同一个函数传不同参数表现不同行为
  - 通常称为overloads
  - Coq中的Adhoc Polymorphism是什么？
    - Notation可以对不同的函数定义同样的符号

多态(Polymorphism)

- Parametric Polymorphism（本节内容）
  - 同一个类别/函数的定义代码对多种类型都适用，给定不同的参数得到不同的具体类型
  - 函数式语言社区通常直接称为多态
  - 面向对象语言社区通常称为泛型
- Subtyping Polymorphism
  - 不同子类的成员函数表现不同的行为
  - 面向对象社区通常直接称为多态
  - 纯函数式语言如Coq通常不支持（高阶函数可以起类似作用）

多态列表

参数X自动加到类型和所有构造函数

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

Check list : Type -> Type.

Check nil : forall X : Type, list X.

Check cons : forall X : Type, X -> list X -> list X.

Check (nil nat) : list nat.

Check (cons nat 3 (nil nat)) : list nat.
```

函数类型，输入类型X，输出List X类型的值。forall 关键字用于引入对X的绑定

多态列表

- 也可以直接写类型来声明list，两种写法等价

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

```
Inductive list: Type -> Type :=
  | nil: forall X: Type, list X
  | cons: forall X: Type, X -> list X -> list X.
```

多态列表

- 或者混用不同写法

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

多态函数

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=
  match count with
  | 0 => nil X
  | S count' => cons X x (repeat X x count')
  end.

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.
```

题外话：依赖类型

- 既然X就是一个参数，能否用nat之类的值而非类型?
  - 可以，这样形成的类型叫做依赖类型，因为依赖一个具体的值
  - 如，长度固定的列表类型：

```
Inductive list' : nat -> Type :=
  | nil'' : list' 0
  | cons'' : forall n, nat -> list' n -> list' (S n).
```

- 依赖类型不在本课程范围

复习：Coq类型推导

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday    => tuesday
  | tuesday   => wednesday
  | wednesday => thursday
  | thursday  => friday
  | friday    => monday
  | saturday  => monday
  | sunday    => monday
  end.
```

复习：Coq类型推导

```
Definition next_weekday' d :=
  match d with
  | monday    => tuesday
  | tuesday   => wednesday
  | wednesday => thursday
  | thursday  => friday
  | friday    => monday
  | saturday  => monday
  | sunday    => monday
  end.
```

多态上的类型推导

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=
  match count with
  | 0 => nil X
  | S count' => cons X x (repeat X x count')
  end.

Fixpoint repeat' X x count : list X :=
  match count with
  | 0        => nil X
  | S count' => cons X x (repeat' X x count')
  end.
```

用下划线的类型推导

- Definition next_weekday' d
- 等价于
- Definition next_weekday’ (d:_) : _

- 用下划线可以省略一些类型实参

```
Fixpoint repeat'' X x count : list X :=
  match count with
  | 0         => nil _
  | S count' => cons _ x (repeat'' _ x count’)
end.
```

- 能否干脆不写类型实参?

隐式参数

```
Arguments nil {X}.
Arguments cons {X}.
Arguments repeat {X}.

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

隐式参数

```
Check nil.
(* nil : forall X : Type, list X *)

Arguments nil {X}.

Check nil.
(* nil : list ?X where ?X : [ |- Type] *)
```

隐式参数

- 类型有可能推不出来，但已经无法给nil传参数

```
Fail Definition mynil := nil.
(* The following term contains unresolved implicit arguments:
  nil
More precisely:
- ?X: Cannot infer the implicit parameter X of nil whose type
is "Type".
*)
Fail Definition mynil := nil nat.
(* Illegal application (Non-functional construction):
The expression "nil" of type "list ?X"
cannot be applied to the term
 "nat" : "Set“
*)
```

Fail确保后面语句出错，
并打印错误消息。

隐式参数

- 可以通过加类型声明帮助推导

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

- 也可以用@临时禁用隐式参数

```
Check @nil : forall X : Type, list X.

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

直接将参数声明为隐式参数

```
Fixpoint repeat''' {X : Type} (x : X) (count : nat) : list X :=
  match count with
  | 0        => nil
  | S count' => cons x (repeat''' x count')
  end.

Check repeat'‘’.
(* repeat''' : ?X -> nat -> list ?X where ?X : [ |- Type] *)
```

直接将参数声明为隐式参数

- 类型的参数也可以是隐式的

```
Inductive list' {X:Type} : Type :=
  | nil'
  | cons' (x : X) (l : list’).

Check list' : Type.
```

- 也可以只把构造函数的参数声明成隐式的

```
Inductive list3 : Type -> Type :=
  | nil3 {X:Type} : list3 X
  | cons3 {X:Type} (x : X) (l : list3 X) : list3 X.

Check list3 : Type -> Type.
```

多态二元组/笛卡尔乘积

```
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.
```

多态二元组/笛卡尔乘积

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

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

多态二元组/笛卡尔乘积

```
Fixpoint combine {X Y : Type} (lx : list X) (ly : list Y)
           : list (X*Y) :=
  match lx, ly with
  | [], _ => []
  | _, [] => []
  | x :: tx, y :: ty => (x, y) :: (combine tx ty)
  end.
```

Combine经常被称作zip

多态option

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

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

多态option

```
Fixpoint nth_error {X : Type} (l : list X) (n : nat)
                   : option X :=
  match l with
  | nil => None
  | a :: l' => match n with
               | O => Some a
               | S n' => nth_error l' n'
               end
  end.
```

高阶函数

- 函数可以作为参数传递
  - 函数式语言的标志性特性
  - 其他语言通常也用不同形式支持
    - C：函数指针
    - 面向对象语言：多态
- Coq的写法

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

Check @doit3times : forall X : Type, (X -> X) -> X -> X.
```

Filter

```
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.
```

Filter

```
Example test_filter1: filter even [1;2;3;4] = [2;4].
Proof. reflexivity. Qed.

Definition length_is_1 {X : Type} (l : list X) : bool :=
  (length l) =? 1.

Example test_filter2:
    filter length_is_1
           [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].
Proof. reflexivity. Qed.

Definition countoddmembers' (l:list nat) : nat :=
  length (filter odd l).
```

匿名函数/λ表达式

- 函数可以用λ表达式声明

```
Example test_anon_fun':
  doit3times (fun n => n * n) 2 = 256.
Proof. reflexivity. Qed.

Example test_filter2':
    filter (fun l => (length l) =? 1)
           [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].
Proof. reflexivity. Qed.
```

Map

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

Map

```
Example test_map1: map (fun x => plus 3 x) [2;0;2] = [5;3;5].
Proof. reflexivity. Qed.

Example test_map2:
  map odd [2;1;2;5] = [false;true;false;true].
Proof. reflexivity. Qed.

Example test_map3:
    map (fun n => [even n;odd n]) [2;1;2;5]
  = [[true;false];[false;true];[true;false];[false;true]].
Proof. reflexivity. Qed.
```

Map

- Map也可以定义在list之外的结构上

```
Definition option_map {X Y : Type} (f : X -> Y) (xo : option X)
                      : option Y :=
  match xo with
  | None => None
  | Some x => Some (f x)
  end.
```

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.
```

Fold

```
Check (fold andb) : list bool -> bool -> bool.

Example fold_example1 :
  fold mult [1;2;3;4] 1 = 24.
Proof. reflexivity. Qed.

Example fold_example2 :
  fold andb [true;true;false;true] true = false.
Proof. reflexivity. Qed.

Example fold_example3 :
  fold app  [[1];[];[2;3];[4]] [] = [1;2;3;4].
Proof. reflexivity. Qed.
```

返回函数

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

Example constfun_example2 : (constfun 5) 99 = 5.
Proof. reflexivity. Qed.

Definition plus3 := plus 3.
Check plus3 : nat -> nat.

Example test_plus3 :    plus3 4 = 7.
Proof. reflexivity. Qed.
Example test_plus3' :   doit3times plus3 0 = 9.
Proof. reflexivity. Qed.
```

作业

- 完成Poly.v中standard非optional且不属于Additional Exercises的6道习题
  - 请使用最新英文版教材

- 如果之前没有接触过函数式语言，可以考虑做一下Additional Exercises中的习题