Chapter 1

Library c04_poly

Capítulo 4 - Polymorphism and Higher-Order Functions (Poly)

```
From LF Require Export c03_lists.
```

Listas polimórficas

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

Função Repetir

```
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)).
```

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

Exercise: 2 stars (mumble_grumble)

```
Inductive mumble : Type :=
  | a : mumble
  | b : mumble → nat → mumble
  | c : mumble.
```

```
Inductive grumble (X : Type) : Type :=
  | d : mumble → grumble X
  | e : X → grumble X.
```

Check d (b a 5).

```
Check d mumble (b a 5).
```

```
Check d bool (b a 5).
```

```
Check e bool true.

Check e mumble (b c 0).
```

Check e bool (b c 0).

```
Check c.
```

Função Repetir

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

Example test_repeat'1 : repeat' nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).

Example test_repeat'2 : repeat' bool false 1 = cons bool false (nil bool).

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

Example test_repeat''1 : repeat'' nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).

Example test_repeat''2 : repeat'' bool false 1 = cons bool false (nil bool).
```

Definições

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

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

Argumentos

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

Função Repetir

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

Listas polimórficas

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

Função Concatenar

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

Função Reverter

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

```
Example test_rev1 : rev (cons 1 (cons 2 nil)) = (cons 2 (cons 1 nil)).
```

```
Example test_rev2: rev (cons true nil) = cons true nil.
```

Função Comprimento

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

```
Example test_length1: length (cons 1 (cons 2 (cons 3 nil))) = 3.
```

Notações

```
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].
```

Exercise: 2 stars, optional (poly_exercises)

```
Theorem app_nil_r : ∀ (X : Type), ∀ l : list X,
  l ++ [] = l.
```

```
Theorem app_assoc : ∀ A (l m n : list A),
  l ++ m ++ n = (l ++ m) ++ n.
```

```
Lemma app_length : ∀ (X : Type) (l1 l2 : list X),
  length (l1 ++ l2) = length l1 + length l2.
```

Exercise: 2 stars, optional (more_poly_exercises)

```
Theorem rev_app_distr : ∀ X (l1 l2 : list X),
  rev (l1 ++ l2) = rev l2 ++ rev l1.
```

```
Theorem rev_involutive : ∀ X : Type, ∀ l : list X,
  rev (rev l) = l.
```

Pares polimórficos

```
Inductive prod X Y :=
| pair : X → Y → prod X Y.
```

Notações

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

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

Projeção x

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

Projeção y

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

Função Combinar

```
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.
```

Exercise: 1 star, optional (combine_checks)

```
Check @combine.
```

Exercise: 2 stars, recommended (split)
Função Dividir

```
Fixpoint split {X Y : Type} (l : list (X×Y)) : (list X) × (list Y) :=
  match l with
  | [] ⇒ ([], [])
  | (x, y) :: t ⇒ let rest := split t in ((x :: fst rest), (y :: snd rest))
  end.
```

```
Example test_split: split [(1,false);(2,false)] = ([1;2],[false;false]).
```

Option

```
Module OptionPlayground.
```

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

End OptionPlayground.
```

Erro

Fixpoint xth_error {X : Type} (l : list X) (x : nat) : option X := match l with | □ => None | a :: l' => if eqb x O then Some a else xth_error l' (pred x) end.

Example test_xth_error1 : xth_error 4;5;6;7 0 = Some 4. Proof. simpl. reflexivity. Qed.

Example test_xth_error2 : xth_error [1];[2] 1 = Some 2. Proof. simpl. reflexivity. Qed.

Example test_xth_error3 : xth_error *true* 2 = None. Proof. simpl. reflexivity. Qed.

Definition hd_error {X : Type} (l : list X) : option X := match l with | □ => None | h :: t => Some h end.

Check @hd_error.

Example test_hd_error1 : hd_error 1;2 = Some 1. Proof. simpl. reflexivity. Qed.

Example test_hd_error2 : hd_error [1];[2] = Some 1. Proof. simpl. reflexivity. Qed.

Funções

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

Check @doit3times.

Example test_doit3times': doit3times negb true = false.

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].

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

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

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

Example test_countoddmembers'1: countoddmembers' [1;0;3;1;4;5] = 4.

Example test_countoddmembers'2: countoddmembers' [0;2;4] = 0.

Example test_countoddmembers'3: countoddmembers' nil = 0.

Example test_anon_fun': doit3times (fun n ⇒ n × n) 2 = 256.

Example test_filter2': filter (fun l ⇒ eqb (length l) 1) [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ] = [ [3];
[4]; [8] ].
```

Exercise: 2 stars (filter_even_gt7)

```
Definition filter_even_gt7 (l : list nat) : list nat := filter (fun n ⇒ andb (evenb n) (negb
(leb n 6))) l.

Example test_filter_even_gt7_1 : filter_even_gt7 [1;2;6;9;10;3;12;8] = [10;12;8].

Example test_filter_even_gt7_2 : filter_even_gt7 [5;2;6;19;129] = [].
```

Exercise: 3 stars (partition)

```
Definition partition {X : Type} (test : X → bool) (l : list X) : list X × list X :=
  (filter test l, filter (fun n ⇒ negb (test n)) l).

Example test_partition1: partition oddb [1;2;3;4;5] = ([1;3;5], [2;4]).

Example test_partition2: partition (fun x ⇒ false) [5;9;0] = ([], [5;9;0]).

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

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].

Example test_map3: map (fun n ⇒ [evenb n;oddb n]) [2;1;2;5] = [[true;false];[false;true];[true;false];[false;tr
```

Exercise: 3 stars (map_rev)

```
Theorem map_list_eq_map_app :
  ∀ (X Y : Type)
    (f : X → Y)
    (l1 l2 : list X),
    map f (l1 ++ l2) = map f l1 ++ map f l2.

Theorem map_rev : ∀ (X Y : Type) (f : X → Y) (l : list X),
  map f (rev l) = rev (map f l).
```

Exercise: 2 stars, recommended (flat_map)

```
Fixpoint flat_map {X Y: Type} (f: X → list Y) (l: list X)
                   : (list Y) :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ f h ++ flat_map f t
  end.

Example test_flat_map1: flat_map (fun n ⇒ [n;n;n]) [1;5;4] = [1; 1; 1; 5; 5; 5; 4; 4; 4].
```

Exercise: 2 stars, optional (implicit_args)

```
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.

Fixpoint map' (X Y : Type) (f : X → Y) (l : list X)
                 : (list Y) :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ f h :: map' _ _ f t
  end.

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.

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].
```

Exercise: 1 star, advanced (fold_types_different)

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

Definition ftrue := constfun true.

Example constfun_example1 : ftrue 0 = true.

Example constfun_example2 : (constfun 5) 99 = 5.

Check plus.

Definition plus3 := plus 3.

Check plus3.

Example test_plus3 : plus3 4 = 7.

Example test_plus3' : doit3times plus3 0 = 9.

Example test_plus3'' : doit3times (plus 3) 0 = 9.
```

Exercise: 2 stars (fold_length)

```
Definition fold_length {X : Type} (l : list X) : nat :=
  fold (fun _ n ⇒ S n) l 0.

Example test_fold_length1 : fold_length [4;7;0] = 3.
```

Prova da corretude de *fold_length*.

```
Theorem fold_length_correct : ∀ X (l : list X),
  fold_length l = length l.
```

Exercise: 3 stars (fold_map)

```
Definition fold_map {X Y: Type} (f: X → Y) (l: list X) : list Y :=
  fold (fun next acc ⇒ f next :: acc) l [].

Theorem fold_map_correct : ∀ (X Y : Type) (f : X → Y) (l : list X),
  fold_map f l = map f l.
```

Exercise: 2 stars, advanced (currying)

```
Definition prod_curry {X Y Z : Type}
  (f : X × Y → Z) (x : X) (y : Y) : Z := f (x, y).

Definition prod_uncurry {X Y Z : Type}
  (f : X → Y → Z) (p : X × Y) : Z := f (fst p) (snd p).

Example test_map1': map (plus 3) [2;0;2] = [5;3;5].

Check @prod_curry.

Check @prod_uncurry.

Theorem uncurry_curry : ∀ (X Y Z : Type) (f : X → Y → Z) x y,
  prod_curry (prod_uncurry f) x y = f x y.

Theorem curry_uncurry : ∀ (X Y Z : Type) (f : (X × Y) → Z) (p : X × Y),
  prod_uncurry (prod_curry f) p = f p.
```

Exercise: 4 stars, advanced (church_numerals)

```
Module Church.

Definition cnat := ∀ X : Type, (X → X) → X → X.

Definition zero : cnat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ x.

Definition one : cnat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ f x.

Definition two : cnat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ f (f x).

Definition three : cnat := @doit3times.
```

Exercise: 1 star, advanced (church_succ)

```
Definition succ (n : cnat) : cnat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ f (n X f x).

Example succ_1 : succ zero = one.

Example succ_2 : succ one = two.
```

Example *succ_3* : *succ two* = *three*.

Example *succ_4* : *succ* (*succ two*) = *succ three*.

Exercise: 1 star, advanced (church_plus)

Definition *plus* (*n m* : *cnat*) : *cnat* :=
  fun (*X* : **Type**) (*f* : *X* → *X*) (*x* : *X*) ⇒ *n X f* (*m X f x*).

Example *plus_1* : *plus zero one* = *one*.

Example *plus_2* : *plus two three* = *plus three two*.

Example *plus_3* : *plus* (*plus two two*) *three* = *plus one* (*plus three three*).

Exercise: 2 stars, advanced (church_mult)

Definition *mult* (*n m* : *cnat*) : *cnat* :=
  fun (*X* : **Type**) (*f* : *X* → *X*) (*x* : *X*) ⇒ *n X* (*m X f*) *x*.

Example *mult_1* : *mult one one* = *one*.

Example *mult_2* : *mult zero* (*plus three three*) = *zero*.

Example *mult_3* : *mult two three* = *plus three three*.

Erro
Definition exp (n m : cnat) : cnat := fun (X : Type) (f : X -> X) (x : X) => (m (X -> X) (fun y => (fun z => (n X y z))) f) x.
Check (exp three two) (@list bool) (map negb).
Example exp_1 : exp two two = plus two two. Proof. reflexivity. Qed.
Example exp_2 : exp three two = plus (mult two (mult two two)) one. Proof. reflexivity. Qed.
Example exp_3 : exp three zero = one. Proof. reflexivity. Qed.

End *Church*.