Logic Foundations
Basics: Functional Programming in Coq

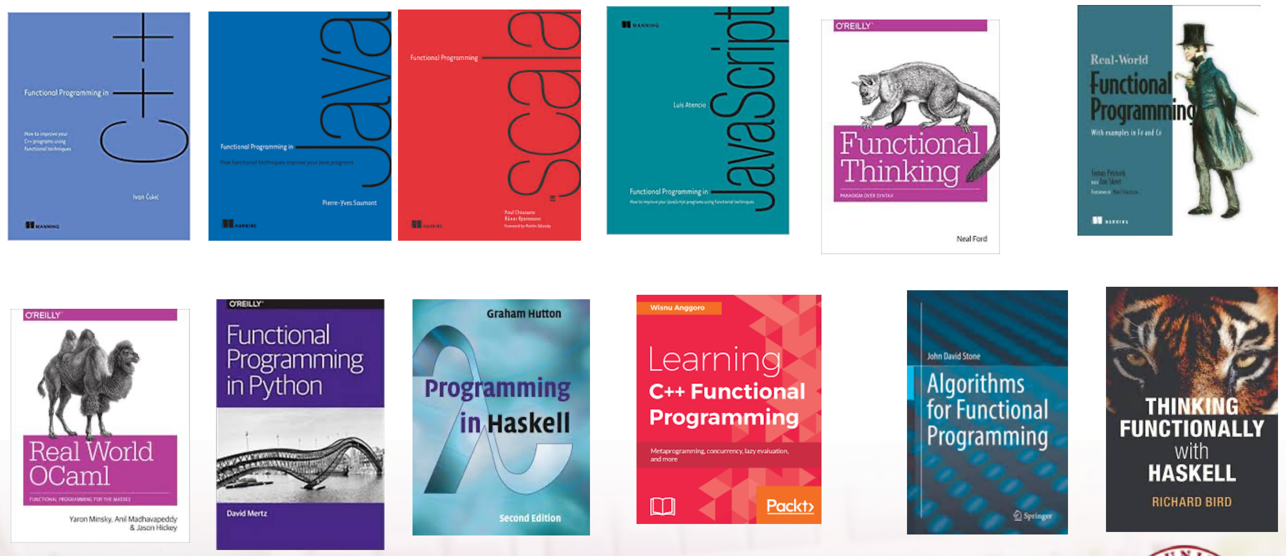
熊英飞 胡振江

信息学院计算机系

2021年3月12日

函数式程序设计

- 纯函数：链接程序和数学对象的纽带
- 高阶函数：函数是可操作的值
- 代数数据类型：易于处理各种数据结构
- 多态类型系统：代码的抽象和重用


Functional Programming in C++
Functional Programming in Java
Functional Programming in Scala
Functional Programming in JavaScript
Functional Thinking
Real-World Functional Programming
Real World OCaml
Functional Programming in Python
Programming in Haskell
Learning C++ Functional Programming
Algorithms for Functional Programming
Thinking Functionally with Haskell

北京大學 PEKING UNIVERSITY

Data and Function

Enumerate Types

Booleans

Numbers

Enumerate Types

Days of the Week

```
Inductive day : Type :=
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  | sunday.
```

Enumerate Types

Function Definition

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

Enumerate Types

Expression Evaluation

```
Compute (next_weekday friday).

Compute (next_weekday (next_weekday saturday)).
```

Enumerate Types

Type Checking

```
Check  next_weekday.
(* next_weekday: day -> day *)

Check  next_weekday
        : day -> day.

Check (next_weekday (next_weekday saturday)).
(* next_weekday (next_weekday saturday) : day *)
```

Enumerate Types

Recording the expected result

```
Example test_next_weekday:
  (next_weekday (next_weekday saturday)) = tuesday.


Proof. simpl. reflexivity.  Qed.
```

Enumerate Types

Code Extraction from Definition

```
Require Extraction.
Extraction Language Scheme.
Extraction next_weekday.
```

```
(define next_weekday (lambda (d)
 (match d
   ((Monday) `(Tuesday))
   ((Tuesday) `(Wednesday))
   ((Wednesday) `(Thursday))
   ((Thursday) `(Friday))
   ((Friday) `(Monday))
   ((Saturday) `(Monday))
   ((Sunday) `(Monday)))))
```

Enumerate Types

Code Extraction from Definition

```
Require Extraction.
Extraction Language OCaml.
Recursive Extraction next_weekday.
```

```
(type day =
| Monday
| Tuesday
| Wednesday
| Thursday
| Friday
| Saturday
| Sunday
```

```
(** val next_weekday : day -> day **)

let next_weekday = function
| Monday -> Tuesday
| Tuesday -> Wednesday
| Wednesday -> Thursday
| Thursday -> Friday
| _ -> Monday
```

Enumerate Types

Code Extraction from Definition

```
Require Extraction.
Extraction Language Haskell.
Recursive Extraction next_weekday.
```

```
module Main where
import qualified Prelude
data Day =
   Monday
 | Tuesday
 | Wednesday
 | Thursday
 | Friday
 | Saturday
 | Sunday
```

```
next_weekday d =
 case d of {
  Monday -> Tuesday;
  Tuesday -> Wednesday;
  Wednesday -> Thursday;
  Thursday -> Friday;
  _ -> Monday}
```

关于作业提交形式

- 对于 .v 不要删除作业，不要改动作业的头尾：

  (** **** Exercise: 1 star, standard (nandb)

  ...

  (** [] *)
- 证明通过的用Qed. 其余的用 Admitted.
- 自我打分：

```
coqc -Q . LF Basics.v
coqc -Q . LF BasicsTest.v
```

Booleans

```
Inductive bool : Type :=
 | true
 | false.
Definition negb (b:bool) : bool :=
 match b with
 | true => false
 | false => true
 end.
Definition andb (b1:bool) (b2:bool) : bool :=
 match b1 with
 | true => b2
 | false => false
 end.
Definition orb (b1:bool) (b2:bool) : bool :=
 match b1 with
 | true => true
 | false => b2
 end.
```

Booleans

```
Example test_orb1: (orb true false) = true.
Proof. simpl. reflexivity. Qed.
Example test_orb2: (orb false false) = false.
Proof. simpl. reflexivity. Qed.
Example test_orb3: (orb false true) = true.
Proof. simpl. reflexivity. Qed.
```

```
Notation "x && y" := (andb x y).
Notation "x || y" := (orb x y).

Example test_orb5:  false || false || true = true.
Proof. simpl. reflexivity. Qed.
```

New Types from Old

```
Inductive rgb : Type :=
  | red
  | green
  | blue.

Inductive color : Type :=
  | black
  | white
  | primary (p : rgb).
```

New Types from Old

```
Definition monochrome (c : color) : bool :=
 match c with
 | black => true
 | white => true
 | primary p => false
 end.

Definition isred (c : color) : bool :=
 match c with
 | black => false
 | white => false
 | primary red => true
 | primary _ => false
 end.
```

Modules

```
Module Playground.
 Definition b : rgb := blue.
End Playground.

Definition b : bool := true.

Check Playground.b : rgb.
Check b : bool.
```

Tuples

```
Inductive bit : Type :=
  | B0
  | B1.

Inductive nybble : Type :=
  | bits (b0 b1 b2 b3 : bit).

Check (bits B1 B0 B1 B0)
    : nybble.
```

Tuples

```
Definition all_zero (nb : nybble) : bool :=
 match nb with
  | (bits B0 B0 B0 B0) => true
  | (bits _ _ _ _) => false
 end.

Compute (all_zero (bits B1 B0 B1 B0)).
(* ===> false : bool *)
Compute (all_zero (bits B0 B0 B0 B0)).
(* ===> true : bool *)
```

Numbers

```
Inductive nat : Type :=
 | O
 | S (n : nat).

Definition pred (n : nat) : nat :=
 match n with
  | O => O
  | S n' => n'
 end.

Definition minustwo (n : nat) : nat :=
 match n with
  | O => O
  | S O => O
  | S (S n') => n'
 end.
```

Numbers

```
Fixpoint evenb (n:nat) : bool :=
 match n with
 | O       => true
 | S O     => false
 | S (S n') => evenb n'
 end.

Fixpoint plus (n : nat) (m : nat) : nat :=
 match n with
  | O => m
  | S n' => S (plus n' m)
 end.

Fixpoint mult (n m : nat) : nat :=
 match n with
  | O => O
  | S n' => plus m (mult n' m)
 end.
```

Numbers

```
Fixpoint minus (n m:nat) : nat :=
 match n, m with
 | O   , _    => O
 | S _ , O    => n
 | S n', S m' => minus n' m'
 end.

Fixpoint exp (base power : nat) : nat :=
 match power with
  | O => S O
  | S p => mult base (exp base p)
 end.
```

Numbers

```
Notation "x + y" := (plus x y)
                    (at level 50, left associativity)
                    : nat_scope.
Notation "x - y" := (minus x y)
                    (at level 50, left associativity)
                    : nat_scope.
Notation "x * y" := (mult x y)
                    (at level 40, left associativity)
                    : nat_scope.
```

Basic Proof Techniques

Proof by Simplification

Proof by Rewriting

Proof by Case Analysis

Proof by Simplification

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
Proof.
  intros n. simpl. reflexivity.  Qed.

Theorem plus_O_n' : forall n : nat, 0 + n = n.
Proof.
  intros n. reflexivity. Qed.

Theorem plus_O_n'' : forall n : nat, 0 + n = n.
Proof.
  intros m. reflexivity. Qed.
```

Proof by Writing

```
Theorem plus_id_example : forall n m:nat,
  n = m ->
  n + n = m + m.

Proof.
  (* move both quantifiers into the context: *)
  intros n m.
  (* move the hypothesis into the context: *)
  intros H.
  (* rewrite the goal using the hypothesis: *)
  rewrite -> H.
  reflexivity.  Qed.
```

Proof by Writing

```
Check mult_n_O.
(* ===> forall n : nat, o = n * o *)

Theorem mult_n_o_m_o : forall p q : nat,
  (p * o) + (q * o) = o.
Proof.
  intros p q.
  rewrite <- mult_n_O.
  rewrite <- mult_n_O.
  reflexivity. Qed.
```

Proof by Case Analysis

```
Fixpoint eqb (n m : nat) : bool :=
 match n with
 | O => match m with
       | O => true
       | S m' => false
       end
 | S n' => match m with
         | O => false
         | S m' => eqb n' m'
         end
 end.

Notation "x =? y" := (eqb x y) (at level 70) : nat_scope.
```

Proof by Case Analysis

```
Theorem plus_1_neq_0_firsttry : forall n : nat,
  (n + 1) =? 0 = false.
Proof.
 intros n. destruct n as [| n'] eqn:E.
  - reflexivity.
  - reflexivity.   Qed.
```

```
Theorem andb_commutative : forall b c, andb b c = andb c b.
Proof.
 intros b c. destruct b eqn:Eb.
 - destruct c eqn:Ec.
   + reflexivity.
   + reflexivity.
 - destruct c eqn:Ec.
   + reflexivity.
   + reflexivity.
Qed.
```

Proof by Case Analysis

```
Theorem plus_1_neq_0_firsttry : forall n : nat,
 (n + 1) =? 0 = false.
Proof.
 intros [|n].
 - reflexivity.
 - reflexivity.   Qed.
```

```
Theorem andb_commutative : forall b c, andb b c = andb c b.
Proof.
  intros [] [].
 - reflexivity.
 - reflexivity.
 - reflexivity.
 - reflexivity.
Qed.
```

Fixpoints and Structural Recursion

```
Fixpoint plus' (n : nat) (m : nat) : nat :=
 match n with
 | O => m
 | S n' => S (plus' n' m)
 end.
```

What this means is that we are performing a **structural recursion over the argument n** -- i.e., that we make recursive calls only on strictly smaller values of n.

作业

- 完成Basics.v中的至少10个练习题。