软件科学基础

Basics: Functional Programming in Coq

熊英飞
北京大学

函数式程序设计语言

- 程序设计语言是对计算过程的描述
- 如何描述计算?
  - 命令式语言
    - 直接描述底层机器应该执行的动作
  - 函数式语言
    - λ演算：计算的本质是函数调用
    - 采用函数调用描述计算
  - 逻辑式语言
    - 用户给出前提、结论和推导式
    - 从前提推导结论的过程就构成了计算

函数式程序设计语言

- 函数式程序设计语言
  - 一直是程序设计语言学术研究的中心
  - 和数学函数对应：更容易理解和掌握，避免副作用带来的Bug
  - 高阶函数：代码更容易复用，也更容易并行化
  - 代数数据类型：更简洁（相对OO的继承）和安全（相对C的Struct和Union）地表示各种数据结构
  - 结构化类型系统：比传统语言的Nominal Type System更强大
  - 函数式数据结构：高效Immutable数据结构实现
- 函数式程序设计语言也是现代程序设计语言的主流
  - 传统语言都在不断集成函数式语言特性
    - Java、C++都在引入Lambda表达式和高阶函数
  - 新程序设计语言通常基于函数式语言设计
    - Scala、Kotlin、Swift等

Coq和Gallina

- Coq是一个交互式定理证明工具
- Gallina是包含在Coq中的描述语言
  - 本身是一个函数式编程语言
  - 同时可以用来描述定理和证明
- Coq采用交互式环境辅助用户编写证明，并分析证明的结果
- 法语词义：
  - Coq：公鸡
  - Gallina：鸟纲鸡形目单词前缀

高卢雄鸡：法国的象征

枚举类型

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

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

函数类型也可以省略

```
Compute (next_weekday friday).
(* = monday : day *)

Compute (next_weekday (next_weekday saturday)).
(* = tuesday : day *)
```

基于空格的函数调用

- 从λ演算继承下来的符号
- f a表示把a作为参数传给f
- 空格是左结合的
  - f a b等价于(f a) b
  - f接收a作为参数，返回另外一个函数，该函数接收b作为参数
  - 实际效果相当于接受两个参数的函数
  - Coq和多数函数式程序设计语言没有接受多个参数的函数
- 空格的优先级通常大于其他操作
  - f a+b表示(f a) + b

定理和证明

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


Proof. simpl. reflexivity.  Qed.
```

Example关键词声明定理，Example可以换成Lemma、Theorem、Fact、Remark等
定理需立刻证明，证明定理采用一系列tactics：
simpl: 化简表达式
reflexivity：根据对称性证明

命题和证明

```
Example test_next_weekday:
  (next_weekday (next_weekday saturday)) = tuesday.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   ========================================
 *   next_weekday (next_weekday saturday) = tuesday
 *)
Proof.
 simpl.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   =================
 *   tuesday = tuesday
 *)
 reflexivity.
(** No more subgoals. *)
  Qed.
```

生成代码

- Coq可生成其他函数语言的代码
  - 目前支持Scheme、Ocaml、Haskell

```
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)))))
```

生成代码

- Coq可生成其他函数语言的代码
  - 目前支持Scheme、Ocaml、Haskell

```
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
```

生成代码

- Coq可生成其他函数语言的代码
  - 目前支持Scheme、Ocaml、Haskell

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

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

定义布尔类型

- Coq有强大的描述能力，基本数据类型都能定义出来
- 本课程我们将重复基础库中的一些定义

```
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.
```

布尔函数使用示例

```
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.
Example test_orb4:  (orb true  true)  = true.
Proof. simpl. reflexivity.  Qed.
```

定义布尔操作的语法

- Coq内嵌了语法定义支持，可以在一定程度下直接引入新语法成分

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

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

If条件表达式

```
Definition negb' (b:bool) : bool :=
  if b then false
  else true.

Definition andb' (b1:bool) (b2:bool) : bool :=
  if b1 then b2
  else false.

Definition orb' (b1:bool) (b2:bool) : bool :=
  if b1 then true
  else b2.
```

- 因为布尔类型不是内置的，所以if可以使用任意包含两个构造函数的类型
- 如果b是由第一个构造函数构造的，则选择第一个分支，否则第二个

类型

```
Check true.
(* ===> true : bool *)

Check true : bool.
Check (negb true) : bool.

Check negb : bool -> bool.

Check orb : bool -> bool -> bool.
```

- Check打印或检查表达式的类型
- bool->bool表示以bool为输入，以bool为输出的函数
- ->右结合，即bool->bool->bool等价于bool->(bool->bool)

带参数的构造函数

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

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

1. 任意两个不同构造函数的值域交集为空
2. 任意构造函数均为单射：两个不同输入产生不同输出

- 提问： color可能有多少不同的值?

使用带参数的构造函数

```
Check black : color.
Check primary : rgb -> color.

Definition monochrome (c : color) : bool :=
  match c with
  | black => true
  | white => true
  | primary p => false (* p匹配任意rgb *)
  end.

Definition isred (c : color) : bool :=
  match c with
  | black => false
  | white => false
  | primary red => true (* 只匹配任意red *)
  | primary _ => false  (* 匹配任意rgb，且不使用该参数 *)
  end.
```

模块系统

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

Definition b : bool := true.

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

- 类似于C++中的namespace和Java中的package

定义多元组

- 带参数的构造函数可直接用于定义多元组

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

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

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

定义多元组

- 带参数的构造函数可直接用于定义多元组

```
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 *)
```

定义自然数

- 自然数的定义：皮亚诺公里
  - 0是自然数
  - 每个自然数有一个后继，后继也是自然数
  - 0不是任何数的后继
  - 如果x和y的后继相同，那么x和y相等
- 用类似的方法定义自然数

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

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

Coq原生自然数类型

- 原生类型采用类似定义，但会自动转换和输出阿拉伯数字

```
Check (S (S (S (S O)))).
(* ===> 4 : nat *)

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

Compute (minustwo 4).
(* ===> 2 : nat *)
```

Coq运算过程

- 需要强调Coq并没有将自然数转换成计算机内部表示并采用相应汇编指令来计算的过程
- Coq的计算过程是在表达式上进行一系列匹配和变换

```
minustwo 2  ⇒  minustwo (S (S 0))  ⇒  match (S (S 0)) with
                                      | O => O
                                      | S O => O
                                      | S (S n') => n'
                                      end
                                                          ⇒  0
```

- 这个特性使得包含某些未知量时也能计算，在证明时非常有用
  - minustwo (S (S n)) = n

定义递归函数

- 采用Fixpoint关键字定义递归函数
- 本课程后期会解释为什么递归函数采用特定关键字

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

练习：定义加法

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

Compute (plus 3 2).
(* ===> 5 : nat *)
```

练习：定义乘法

等价于(n:nat) (m:nat)

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

定义乘方

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

定义减法

- match可同时匹配多个参数

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

结构递归
Structural Recursion

- Coq要求所有的递归都必须是终止的
  - 否则类型检查可能会不终止

- Coq通过要求所有递归是结构递归来保证这一点
  - 结构递归：递归调用的实参是原形参的子结构
  - 结构递归一定终止
  - 非结构递归不一定不终止

终止的非结构递归

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

- 为什么不是结构递归?

Coq的分析能力有限

```
Fixpoint quicksort (l: list nat) :=
  match l with
    | nil => nil
    | h :: t => (quicksort (filter_lte h t))
                  ++ h :: (quicksort (filter_gt h t))
  end.
```

- Coq无法推出来filter_xxx函数返回的是t的子部分，会执行报错

定义四则运算运算符

```
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.
Check ((0 + 1) + 1) : nat.
```

- at level n： n越小，优先级越高
- left/right associativity： 左结合/右结合
- nat_scope： 如果Coq混淆了多种语法定义，可以用(x*y)%nat来强制使用nat_scope
- Notation写法本课程不要求掌握

自然数的比较

```
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.
```

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

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

加上问号和命题中使用的=和<=区别
这里是返回用户定义类型bool的函数

带全称量词的命题

```
Theorem plus_O_n : forall n : nat, 0 + n = n.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   ============================
 *   forall n : nat, 0 + n = n
 *)
Proof. intros n.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   n : nat
 *   =========
 *   0 + n = n
 *)
```

- intros n：假设n为某个任意的自然数，将n移入假设区域，表示“当前能用的值”
- n也可以省略或换成m

带全称量词的命题

```
 simpl.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   n : nat
 *   =======
 *   n = n
 *)
 reflexivity.
(** No more subgoals. *)
  Qed.
```

如果原定理是n+0=n，还能用这种方式证明吗？

带全称量词的命题

```
Theorem plus_O_n'' : forall n : nat, 0 + n = n.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   =========================
 *   forall n : nat, 0 + n = n
 *)
Proof. intros m.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   m : nat
 *   =========
 *   0 + m = m
 *)
 reflexivity.
(** No more subgoals. *)
 Qed.
```

reflexivity会自动应用simpl

通过Rewrite证明

```
Theorem plus_id_example : forall n m:nat,
  n = m ->
  n + n = m + m.
Proof.
  intros n m. intros H.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   n, m : nat
 *   H : n = m
 *   =============
 *   n + n = m + m
 *)
  (* rewrite the goal using the hypothesis: *)
```

- 移入H到假设区域，表示“当前能用的假设”
- 之后会学到“值”和“假设”本质一样

通过Rewrite证明

```
  rewrite -> H.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   n, m : nat
 *   H : n = m
 *   =============
 *   m + m = m + m
 *)
  reflexivity.(** No more subgoals. *)
  Qed.
```

将目标中H的左边都替换成右边。H需要为等式，可以是假设中的等式，也可以是之前证明的定理。

通过Rewrite证明

```
  rewrite <- H.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   n, m : nat
 *   H : n = m
 *   =============
 *   n + n = n + n
 *)
  reflexivity.(** No more subgoals. *)
  Qed.
```

将目标中H的右边都替换成左边。H需要为等式，可以是假设中的等式，也可以是之前证明的定理。箭头也可省略，默认朝右

输出定理

- Check也可以用于输出定理定义

```
Check plus_id_example.
(**
 * plus_id_example
 *      : forall n m : nat, n = m  -> n + n = m + m
 *)
```

- 之后会学到，定理定义和类型是等价的

通过分类讨论证明

```
Theorem plus_1_neq_0 : forall n : nat,
  (n + 1) =? 0 = false.
Proof.
  intros n. destruct n as [| n'] eqn:E.
  - reflexivity.
  - reflexivity.   Qed.
```

- destruct n： 将n按类型的归纳定义分解
  - as [| n’]： 可省略，给定义参数取名
  - eqn:E： 保留分解之前和之后的等价关系作为假设E，如省略则不保留
- “-”： 标记证明子目标中一项，如省略则按序证明

其他证明排版方法

```
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.
```

- +、-、*的任意多次重复都可以标记子目标
  - 如+++或**

其他证明排版方法

```
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.
```

- {} 标记一个子目标的证明
- {}和+, -, *可以随意混合使用

作业

- 完成Bascis.v中standard非optional的习题，More Exercise下的Course Late Policies除外，共11道
    - 请使用最新英文版教材