

如何写出没有Bug的程序？
——形式化证明介绍

熊英飞

北京大学

程序出错可能导致灾难性事故

2019年波音737Max坠机事件：埃塞俄比亚航空一架波音737 MAX 8型飞机在起飞阶段坠毁，机上人员全数遇难。



事故原因：飞机MCAS防失速自动系统软件存在缺陷

2016年特斯拉车祸：自动驾驶模式下的特斯拉汽车和卡车相撞，导致驾驶员当场丧生



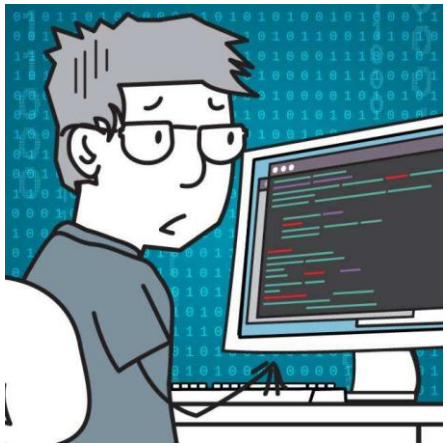
事故原因：在强烈日光条件下，摄像头进入盲区，但软件系统并没有捕获这一情况

2011年亚马逊宕机事故：亚马逊云计算出现了超过2天的宕机事故，造成的资金和信誉损失难以估算



事故原因：软件配置错误导致部分结点请求激增，不断转发请求压垮网络

如何知道程序是正确的?



程序员

给我写一个排序

写好了，看:

```
quicksort (x:xs) =
  quicksort [a | a <- xs, a <= x]  ++ [x]
  ++ quicksort [a | a <- xs, a > x]
```

写对了吗?

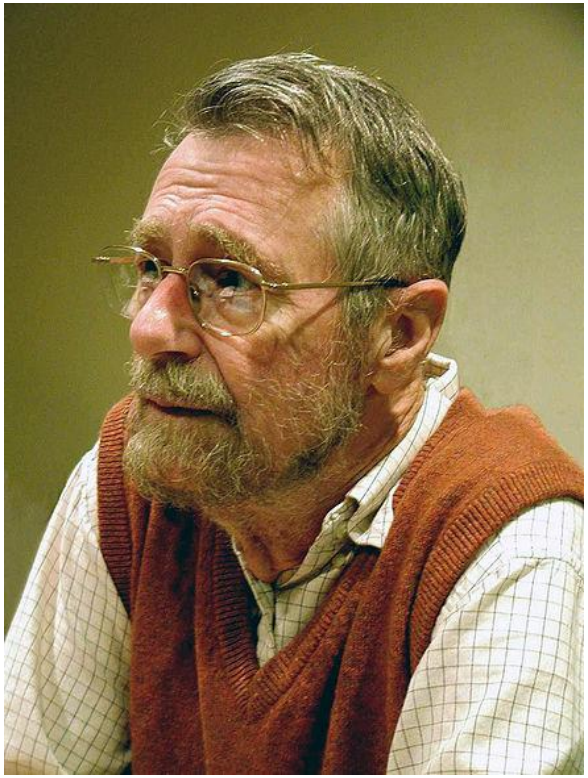


老板

如何知道程序是正确的?

-  程序员：我测试了!

Testing shows the presence, not the absence of bugs.



Edsger W. Dijkstra

如何知道程序是正确的?



- 程序员：你看，我先把比x小的选出来排序，然后把比x大的选出来排序，然后把三部分按顺序合起来，过程多么合理!
  - 自然语言存在二义性
    - 资深程序员：“排序”的要求是什么？升序还是降序？要求时间复杂度吗？排序元素的定义域和序列长度通常哪个大?
    - 数学家：“排序”的定义是什么？被排序元素上存在全序吗？存在偏序吗?
  - 不容易判断自然语言的分析是否全面、正确
    - 上述描述没有分析终止性和边界条件

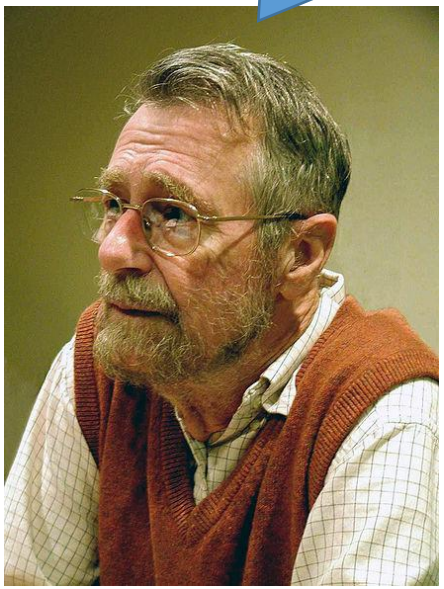
“牌序”？好像是挺重要的



如何解决自然语言论证的问题?

- 数学：对事物的抽象结构与模式进行严格描述的一种通用手段（百度百科）
  - 解决二义性问题
- 逻辑学：研究推理的学科，即研究如何从前提必然推出结论（百度百科）
  - 解决论证正确性问题
- 数理逻辑：严格描述的逻辑学，是现代数学的基础
  - 一系列语法规则，用于描述命题
  - 一系列推理规则，用于证明命题是否成立

假如我早年在数理逻辑上好好下点功夫的话，我就不会犯这么多的错误。不少东西逻辑学家早就说了，可我不知道。要是我能够年轻20岁的话，就要回去学逻辑。



Edsger W. Dijkstra

数理逻辑历史



- 逻辑学的发源一般追溯到亚里斯多德
  - 古希腊三圣：苏格拉底、柏拉图、亚里斯多德
  - 一般也被认为是科学的发源
- 19世纪数学家们试图给数学建立基础理论
  - 1854年，英国数学家布尔(George Boole)提出布尔代数
    - 注意：没有boolean这个词，只有Boolean
    - 约等于命题逻辑
  - 1879年，德国数学家戈弗雷格(Gottlob Frege)引入量词
  - 1913年，英国数学家怀特海(Alfred North Whitehead)和罗素(Bertrand Russell)引入了公理和推导规则

形式系统

- 逻辑通过形式系统来定义
  - 现代数学中，逻辑、演算、代数都是同义词，表示某种形式系统，不同领域根据习惯采用不同词
- 形式系统包括以下四个部分
  - 字母表Alphabet：一个符号的集合Σ
  - 文法Grammar：一组文法规则，定义Σ^*的一个子集，为该形式系统中合法命题集合
  - 公理模式Axiom Schemata：一组公理模板，通过字符串代换，可以产生为真的命题
    - 如：对任意命题P, Q，$P \wedge Q \rightarrow P$
  - 推导规则Inference Rules：一组推导规则，通过字符串匹配和替换，用于推导出公理以外为真的命题
    - 如：对任意命题P, Q，$P \wedge Q \Rightarrow P$

如何用数理逻辑来论证程序正确性?



- 程序员：用数理逻辑证明如下定理：
  - $\forall l, i\colon bag(l) = bag\big(quicksort(l)\big) \wedge quicksort(l)[i-1] \leq quicksort(l)[i]$
  - 证明：
    - 根据XX公理，得到XX
    - 根据XX定理，得到XX
    - ……
- 问题：怎么知道证明写对了
  - https://www.win.tue.nl/~gwoegi/P-versus-NP.htm
    - 至少有62篇论文证明了$P = NP$，50篇论文证明了$P \neq NP$

能否自动检查证明的正确性?

- 能，并且能套用编译器的类型检查算法



“‘命题-证明’和‘类型-值’之间存在对应关系。”
——Curry-Howard Correspondence，1934-1969

交互式定理证明工具



- 本讲座：交互式定理证明工具Rocq
  - 全球使用最广的交互式定理证明工具之一
  - 包含函数式编程语言Gallina
  - 支持基于Gallina编写的程序定义命题
  - 支持采用数理逻辑撰写证明
  - 自动检查证明是否证明命题
    - 并随时提示程序员还没有完成证明的部分
- 法语词义：
  - Coq(Rocq原名)：公鸡
  - Gallina：鸟纲鸡形目单词前缀

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

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
  - Rocq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.
(** [Rocq Proof View]
 * 1 subgoal
 *
 *   ============================================
 *   next_weekday (next_weekday saturday) = tuesday
 *)
Proof.
 simpl.
(** [Rocq Proof View]
 * 1 subgoal
 *
 *   =================
 *   tuesday = tuesday
 *)
 reflexivity.
(** No more subgoals. *)
  Qed.
```

生成代码

- Rocq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}
```

定义布尔类型



- Rocq有强大的描述能力，基本数据类型都能定义出来

```
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



- Rocq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?

模式匹配：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.
```

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.
```

Rocq原生自然数类型



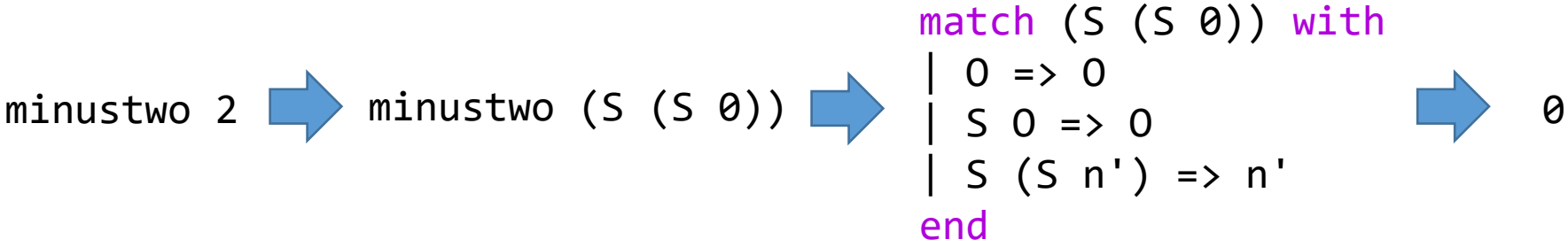
- 原生类型采用类似定义，但会自动转换和输出阿拉伯数字

```
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 *)
```

Rocq运算过程



- 需要强调Rocq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
- Rocq的计算过程是在表达式上进行一系列匹配和变换



- 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

```
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.
```

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.
(** [Rocq Proof View]
 * 1 subgoal
 *
 *   ============================
 *   forall n : nat, 0 + n = n
 *)
Proof. intros n.
(** [Rocq 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.
(** [Rocq 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？

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.
(** [Rocq 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”
- 在Rocq内部“值”和“假设”本质一样

通过Rewrite证明




```
  rewrite -> H.
(** [Rocq 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.
(** [Rocq 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

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

不能通过simpl和reflexivity证明n+0=n




```
Theorem add_0_r_firsttry : forall n:nat,
  n + 0 = n.
Proof. intros n.
(*   n : nat
 *   =========
 *   n + 0 = n
 *)
  simpl.(** [Rocq Proof View]
 *   n : nat
 *   =========
 *   n + 0 = n
 *)
 (* Does nothing! *)
Abort.
```

Destruct也不行




```
Theorem add_0_r_secondtry : forall n:nat,
  n + 0 = n.
Proof.
  intros n. destruct n as [| n'] eqn:E.
  - (* n = 0 *)
    reflexivity. (* so far so good... *)
  - (* n = S n’ *)
    (* S n' + 0 = S n’ *)
    simpl.
    (* S (n' + 0) = S n’ *)
Abort.
```

结构归纳法
Structural Induction



- 数学归纳法：
  - 首先证明性质P对0成立
  - 然后证明P(k)->P(k+1)
- 结构归纳法：
  - 对于任意递归定义的类型
  - 证明P对该类型的每一个构造函数都成立
    - 如果构造函数接受该类型的参数，则假设P对参数成立
- 数学归纳法是结构归纳法在自然数上的特例

采用结构归纳法证明




```
Theorem add_0_r : forall n:nat, n + 0 = n.
(** [Rocq Proof View]
 * 1 subgoal
 *
 *   ============================
 *   forall n : nat, n + 0 = n
 *)
Proof. intros n.
(** [Rocq Proof View]
 * 1 subgoal
 *
 *   n : nat
 *   =========
 *   n + 0 = n
 *)
```

采用结构归纳法证明



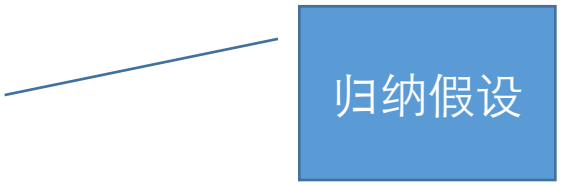

```
 induction n as [| n' IHn'].
(** [Rocq Proof View]
 * 2 subgoals
 *
 *   ==============
 *   0 + 0 = 0
 *
 * subgoal 2 is:
 *  S n' + 0 = S n'
 *)
```

为变量命名，可省略

采用结构归纳法证明




```
  - reflexivity.
  - (** [Rocq Proof View]
 * 1 subgoal
 *
 *   n' : nat
 *   IHn' : n' + 0 = n'
 *   ==================
 *   S n' + 0 = S n'
 *)
    simpl. rewrite -> IHn'. reflexivity.
Qed.
```



Induction同时作用于一个变量的多个实例




```
Theorem minus_n_n : forall n,
  minus n n = 0.
Proof.
  (* WORKED IN CLASS *)
  intros n. induction n as [| n' IHn'].
  - (* n = 0 *)
    simpl. reflexivity.
  - (* n = S n' *)
    simpl. rewrite -> IHn'. reflexivity.  Qed.
```

证明子定理




```
Theorem mult_0_plus' : forall n m : nat,
  (0 + n) * m = n * m.
Proof.
  intros n m.
  assert (H: 0 + n = n). { reflexivity. }
  rewrite -> H.
  reflexivity.  Qed.
```

证明子定理




```
Theorem mult_0_plus' : forall n m : nat,
  (0 + n) * m = n * m.
Proof. intros n m.
(** [Rocq Proof View]
 * 1 subgoal
 *
 *   n, m : nat
 *   ==================
 *   (0 + n) * m = n * m
 *)
```

证明子定理




```
  assert (H: 0 + n = n).
(** [Rocq Proof View]
 * 2 subgoals
 *
 *   n, m : nat
 *   ==================
 *   0 + n = n
 *
 * subgoal 2 is:
 *  (0 + n) * m = n * m
 *)
```

证明子定理




```
{ reflexivity. }
 (** [Rocq Proof View]
* 1 subgoal
*
*   n, m : nat
*   H : 0 + n = n
*   ============================
*   (0 + n) * m = n * m
*)
rewrite -> H.
reflexivity.  Qed.
```

Lists




```
Inductive natlist : Type :=
  | nil
  | cons (n : nat) (l : natlist).

Definition mylist := cons 1 (cons 2 (cons 3 nil)).

Notation "x :: l" := (cons x l)
                     (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).

Definition mylist1 := 1 :: (2 :: (3 :: nil)).
Definition mylist2 := 1 :: 2 :: 3 :: nil.
Definition mylist3 := [1;2;3].
Definition mylist4 := 1 :: 1 + 1 :: 3 :: nil.
```

常用函数




```
Fixpoint repeat (n count : nat) :
natlist :=
  match count with
  | O => nil
  | S count' => n :: (repeat n count')
  end.

Fixpoint length (l:natlist) : nat :=
  match l with
  | nil => O
  | h :: t => S (length t)
  end.
```

常用函数




```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil    => l2
  | h :: t => h :: (app t l2)
  end.

Notation "x ++ y" := (app x y)
                     (right associativity, at level 60).

Example test_app1:             [1;2;3] ++ [4;5] = [1;2;3;4;5].
Proof. reflexivity. Qed.
Example test_app2:             nil ++ [4;5] = [4;5].
Proof. reflexivity. Qed.
Example test_app3:             [1;2;3] ++ nil = [1;2;3].
Proof. reflexivity. Qed.
```

常用函数




```
Definition hd (default : nat) (l : natlist) : nat :=
  match l with
  | nil => default
  | h :: t => h
  end.

Definition tl (l : natlist) : natlist :=
  match l with
  | nil => nil
  | h :: t => t
  end.

Example test_hd1:             hd 0 [1;2;3] = 1.
Proof. reflexivity. Qed.
Example test_hd2:             hd 0 [] = 0.
Proof. reflexivity. Qed.
Example test_tl:              tl [1;2;3] = [2;3].
Proof. reflexivity. Qed.
```

证明列表的性质




```
Theorem nil_app : forall l : natlist,
  [] ++ l = l.
Proof. reflexivity. Qed.

Theorem tl_length_pred : forall l:natlist,
  pred (length l) = length (tl l).
Proof.
  intros l. destruct l as [| n l'].
  - (* l = nil *)
    reflexivity.
  - (* l = cons n l' *)
    (* pred (length (n :: l')) = length (tl (n :: l')) *)
    simpl.
    (* length l' = length l' *)
    reflexivity.  Qed.
```

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
  - Rocq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Rocq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.
```

隐式参数




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

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

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] *)
```

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
    - 参数为类型的情况
  - 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).
```

Options：处理例外情况



- 图灵奖Tony Hoare：我发明Null是一个错误，造成十亿美元的损失
- Null的问题：不处理null值编译器也不报警
- 如何让编译器报警?

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

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

Options：处理例外情况




```
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
- Rocq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.
```

小结



- 本讲座介绍了基础的函数式编程和基础的证明方法
  - 已经覆盖了Rocq的基础语言特性
  - 其他高级特性都可以用基础语言特性定义出来
    - A->B
      - 表示以A类型的值为输入，返回B类型的值的函数
      - 也表示A蕴含B：A命题的证明为输入，返回B命题的证明的函数
    - and

```
Inductive and (P Q : Prop) : Prop :=
| conj : P -> Q -> and P Q.
```

但



- 为什么很少听说有Rocq编写的软件系统?

结构递归
Structural Recursion



- Rocq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

- Rocq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?

Rocq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.
```

- Rocq无法推出来filter_xxx函数返回的是t的子部分，会执行报错

实际中如何用Rocq证明程序的正确性?



- 在Rocq中定义数据类型表示C语言的语法
  - 形式语法：定义程序的结构和合法性
- 在Rocq中定义公理表示C语言的语义
  - 形式语义：定义程序在运行的时候会做什么
- 将程序解析成语法数据结构后，基于语义公理证明程序有关的定理
- 著名软件系统:
  - seL4：证明完全正确的操作系统内核
  - CompCert：证明完全正确的C编译器

更进一步



- 之前的证明方式并不是很方便
  - 霍尔逻辑：专门针对证明程序正确性设计的逻辑系统
  - 分离逻辑：专门针对指针正确性设计的逻辑系统
- 相关工具
  - VST/QCP（上海交大）
    - 基于Rocq，对C语言添加了霍尔逻辑和分离逻辑支持
    - https://ide.qua.codes/
  - C*（北京大学）
    - 重新基于C设计了命令式证明语言

霍尔三元组



- {前条件}语句{后条件}
- 如
  - $\{x > 0\}$ x := x + 5 $\{x > 5\}$
  - $\{x > 0\}$ x := x + 5 $\{x > 0\}$
  - $\{x = n \wedge y \neq 0\}$ x := x / y $\{x * y = n\}$
  - $\{True\}$ while(true) x := x + 1 $\{False\}$

- 含义：如果程序状态满足前条件，执行语句后，状态一定满足后条件

霍尔逻辑规则

$$\textsc{Skip}\ \frac{}{\{P\}\ \textbf{skip}\ \{P\}} \qquad \textsc{Assign}\ \frac{}{\{P[a/x]\}\ x := a\ \{P\}}$$

$$\textsc{Seq}\ \frac{\{P\}\ c_1\ \{R\} \qquad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1; c_2\ \{Q\}} \qquad \textsc{If}\ \frac{\{P \wedge b\}\ c_1\ \{Q\} \qquad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

$$\textsc{Consequence}\ \frac{\vDash (P \Rightarrow P') \qquad \{P'\}\ c\ \{Q'\} \qquad \vDash (Q' \Rightarrow Q)}{\{P\}\ c\ \{Q\}}$$

$$\textsc{While}\ \frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{P \wedge \neg b\}}$$

用霍尔逻辑证明举例



- if (x > 0) x := x+10 else x := 20
  - 该程序执行结束后，x是否一定大于0?
- 根据Assign，可得
  - {x+10>0} x := x+ 10 {x > 0}
  - {True} x := 20 {x >0}
- 因为x>0 => x+10 > 0且¬x>0 => True，根据Consequence，可得
  - {x>0} x := x+10 {x > 0}
  - {¬x>0} x := 20 {x > 0}
- 根据If，可得
  - {True} if (x > 0) x := x+10 else x := 20 {x>0}

用霍尔逻辑证明练习



- while (x < 10) x := x+1
  - 该程序执行结束后，x是否一定大于0?
- 根据Assign，可得
  - {True} x := x+1 {True}
- 根据Consequence，可得
  - {x<10/\True} x := x+1 {True}
- 根据While，可得
  - {True} while (x < 10) x += 1; {True /\ x>=10}
- 根据Consequence，可得
  - {True} while (x < 10) x += 1; {x>0}

更进一步



- 基于这些工具的证明代价仍然很大
  - SMT：一组自动证明一阶逻辑命题的算法
- 相关工具：
  - CoqHammer
    - 集成SMT求解器到Rocq中，自动证明Rocq定理
    - https://coqhammer.github.io/
  - ReCent-Prover（北京大学）
    - 集成CoqHammer和大模型的自动证明Rocq定理
    - https://github.com/gtwgtwgtw/Proof-agent/

后续学习



- 北京大学《软件科学基础课》
  - https://xiongyingfei.github.io/SF/
  - 包括本讲座的内容和其他程序设计语言基础内容
    - Rocq程序设计、数理逻辑、形式语义、霍尔逻辑、类型系统等
- 北京大学《软件分析技术》
  - https://xiongyingfei.github.io/SA_new/
  - 包括自动证明程序正确性的各种技术
    - 基于抽象解释的程序分析、基于SMT求解器的程序验证等



谢谢！