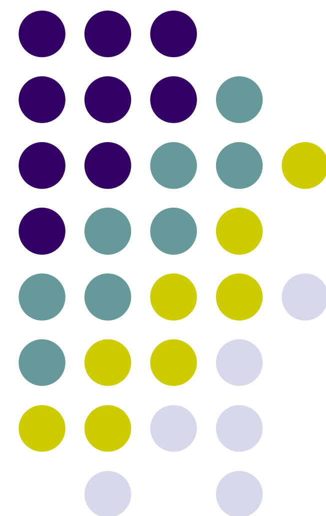
南京大学
Nanjing University

第九讲- coq的高阶特性与策略



内容提要

- 多态与结构化数据
  - 多态 | 隐式参数机制
- 高阶函数
  - `filter` | `map` | `fold`
- 高阶策略
  - `injection` | `discriminate` | `apply` | `Induction`

Part1- 多态与结构化数据

范式的转变：过程式编程

- **Programs = Algorithms + Data structure**

- 数据结构：如何高效布局和管理内存状态
  - 内存中一块被命名的区域
  - 可以存储数据本身，也可以存储数据的信息
  - 可以被算法修改自身状态来推进程序的执行过程

```
struct Node {
    int data;           // 数据负载
    struct Node* next;  // 指向下一块内存的指针
};
// 算法: 修改 next 指针来插入元素
```

范式的转变：函数式编程

- Proof = Types + Functions
  - Proof as 程序
  - Types as 结构化数据
  - Functions as 类型匹配

- 结构化数据：如何为证明中定义其所需要公理和规则
  - 数据的逻辑形态，而非内存状态
  - 不可被修改，只能构造新的数据

```
Inductive natlist : Type :=
  | nil_n : natlist                          (* 形状1: 空 *)
  | cons_n : nat -> natlist -> natlist. (* 形状2: 头接尾 *)
```

Curry-Howard Correspondence

- 程序员视角
  - 结构化数据：定义数据的形状
  - 函数：定义数据的变换
  - 程序：数据变换的组合

- 逻辑学家视角
  - 结构化数据：定义公理和推理规则
  - 函数：描述归纳推理的过程
    - `Match`：分类讨论
    - `Fixpoint`：归纳法
  - 程序：最顶层的函数

自然数链表 natlist

```
Inductive natlist : Type :=
  | nil_n : natlist                              (* 形状1：空 *)
  | cons_n : nat -> natlist -> natlist. (* 形状2：头接尾 *)
```

- **Inductive natlist : Type**
  - 定义一个名字叫**natlist**的归纳类型，类型是**Type**
- **nil_n : natlist**
  - 第一种构造情况，一个空的链表
- **cons_n : nat -> natlist ① -> natlist ②**
  - 第二种构造情况，一个自然数**nat**加上一个自然数链表 ① **natlist**，组成一个新的自然数链表**natlist** ②

布尔值 boollist

```
Inductive boollist : Type :=
  | nil_b : boollist
  | cons_b : bool -> boollist -> boollist.
```

- natlist和boollist具有完全相同的逻辑结构
  - 区别仅在于归纳构造过程中是加入nat还是bool

- 过程式编程的应对方式：使用通用指针void*或基础类型对象object
- 函数式编程的应对方式：**将数据的结构与内容进行分离**

泛型编程：抽象的结构

```
(* list 是一个纯粹的结构模板 *)
Inductive list (X : Type) : Type :=
  | nil : list X
  | cons : X -> list X -> list X.
```

- 多态 **Polymorphism**：
  - 将“内容的类型”作为数据的类型参数
  - 保留纯粹结构描述
- 证明（程序）的复用：
  - **list**描述了一种类型的逻辑结构
  - 如果证明了**list**具有某种性质，则这种性质对于**natlist**和**boollist**都成立

多态结构的实例化

```
(* 这是一个包含元素 1 的自然数列表 *)
Definition my_nat_list : list nat :=
  cons nat 1 (nil nat).
```

链表中的元素1和空链表的类型都是nat

- 定义抽象列表具体的类型
  - 类型为nat
- 定义列表中的数据
  - 使用构造子cons构造一个类型是nat的列表
  - 1: cons构造子的参数 X
  - (nil nat): cons构造子的第二个参数 list X
    - 使用构造子nil构造了一个类型是nat的列表

使用隐式参数

- 在定义时使用大括号{ }代替圆括号( )

```
Inductive list {X : Type} : Type := ...
```

- 可以像使用**natlist**或**boollist**一样定义多态链表了
  - coq可以基于参数本身的类型推断出类型参数**X**

```
Check (cons 1 (cons 2 nil)).   (* Coq 推断 X=nat *)
Check (cons true nil).         (* Coq 推断 X=bool *)
```

构造子的符号描述

- 使用 :: 描述构造子cons

```
Notation "x :: y" := (cons x y)
                     (at level 60, right associativity).
```

  - 1: : 2: : nil等价于cons 1 (cons 2 nil)

- 使用 [ ] 描述实例化的声明

```
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
```

  - [1;2;3]等价于1::2::3::nil

函数中的多态

- 编写函数**repeat**，创建一个将元素**x**重复**count**次的**list**

```
(* 注意: X 是隐式参数，但在函数定义头中通常显式写出 {X:Type} *)
Fixpoint repeat {X : Type} (x : X) (count : nat) : list X :=
  match count with
  | O => nil
  | S count' => x :: (repeat x count')
  end.
```

```
Compute (repeat 5 3).
(* 结果: [5; 5; 5] : list nat *)

Compute (repeat true 2).
(* 结果: [true; true] : list bool *)
```

更多的多态函数

- 统计list的长度

```
Fixpoint length {X : Type} (l : list X) : nat :=
  match l with
  | nil => 0
  | h :: t => S (length t)
  end.
```

- 拼接两个list

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

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

更多多态结构：积

```
Inductive prod (X Y : Type) : Type :=
  | pair : X -> Y -> prod X Y.

Arguments pair {X} {Y} _ _.      (* 隐式参数设置 *)
Notation "( x , y )" := (pair x y).
Notation "X * Y" := (prod X Y) : type_scope.
```

- X和Y是两个独立的类型参数
- 应用场景
  - 同时返回两个返回值
  - 在证明中同构于 A /\ B

更多多态结构: 可选项

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

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

- 互补的两种情况
  - **Some**：存在一个参数，类型为**X**
  - **None**：不存在参数
- 应用场景
  - 异常处理：**error : nat -> list X -> option X**
    - 访问X类型的链表的第**nat**位
    - 如果正常访问，则返回对应值（**Some**）
    - 如果数组越界，则返回**None**

Part2- 高阶函数

向列表[1;2]末尾添加元素3

```
void append(List* list, int val) {
    Node* current = list->head;
    while(current->next != NULL) {
        current = current->next; // 1. 遍历
    }
    // 2. 修改：将最后一个节点的指针指向新节点
    current->next = new Node(val);
}
```

- 过程式语言中函数的特征
  - **副作用：**修改后的结果不依赖于返回值返回，而是直接作用于原有的`list`
  - **破坏性：**完成计算后，原有的`list[1;2]`不再存在

向列表[1;2]末尾添加元素3

```
Fixpoint app (l : list nat) (val : nat) : list nat :=
  match l with
  | nil => [val]
  | h :: t => h :: (app t val) (* 递归构造 *)
  end.
```

用h和app t val的返回值生成一个新的list

- 函数式语言中函数的特征
  - **纯函数：**修改后的结果依赖于返回值返回，并不直接作用于原有的list
  - **持久性：**完成计算后，原有的list[1;2]依然存在

coq的标准库函数

- Coq.Lists.List包含了数十种常用的列表操作函数
  - filter: 筛选
  - map: 映射
  - fold: 规约

```
Require Import Coq.Lists.List.
Import ListNotations.  (* 开启像 [1; 2] 这样的符号糖 *)
```

- 其他标准库
  - Coq.Arith.Arith | Coq.ZArith.ZArith| Coq.Reals.Reals
  - Coq.Strings.String | Coq.Bool.Bool
  - Coq.Logic.Classical

筛选函数 filter

- 目标：对结构化数据中的元素进行筛选
  - 用户自定义筛选逻辑，并作为函数参数传递给**filter**

```
Fixpoint filter {X : Type} (test : X -> bool) (l : list X)
                : list X :=
  match l with
  | nil => nil  (* Base case: 空表过滤后还是空表 *)
  | h :: t =>
      if test h
      then h :: (filter test t)  (* 通过：留下来，接上后面的结果 *)
      else filter test t         (* 失败：扔掉，直接处理后面的 *)
  end.
```

  - **test : X -> bool**：用户的筛选逻辑

筛选函数 filter

- 定义筛选逻辑

```
(* 定义判断偶数的函数 *)
Fixpoint evenb (n:nat) : bool :=
  match n with
  | O        => true
  | S O      => false
  | S (S n') => evenb n'
  end.

(* 定义判断奇数的函数 (即“非偶数”) *)
Definition oddb (n:nat) : bool := negb (evenb n).
```

- 保留奇数

```
Compute filter oddb [1; 2; 3; 4; 5].
(* Coq 输出结果:
   = [1; 3; 5]
   : list nat
*)
```

- 保留偶数

```
Compute filter evenb [1; 2; 3; 4; 5].
(* Coq 输出结果:
   = [2; 4]
   : list nat
*)
```

映射函数 map

- 目标：对结构化数据的元素进行统一操作
  - 用户自定义操作逻辑，并作为函数参数传递给map

```
Fixpoint map {X Y : Type} (f : X -> Y) (l : list X)
               : list Y :=
  match l with
  | nil => nil (* Base case: 空表没东西可变 *)
  | h :: t => (f h) :: (map f t) (* 核心: 变换头元素，递归变换尾部 *)
  end.
```

  - **f : X -> Y:** 用户的操作逻辑

映射函数 map

- 数值变换，所有元素统一乘以10

```
Compute map (fun x => x * 10) [1; 2; 3].
(* Coq 输出结果:
   = [10; 20; 30]
   : list nat
*)
```

- 类型变换，判断每一个元素是否为偶数

```
Compute map evenb [1; 2; 3; 4].
(* Coq 输出结果:
   = [false; true; false; true]
   : list bool
*)
```

规约函数 fold

- 目标：对结构化数据的结构本身进行变换
  - 用户自定义结构变换逻辑，并作为函数参数传递给fold

```
Fixpoint fold {X Y : Type} (f : X -> Y -> Y) (l : list X) (b : Y)
              : Y :=
  match l with
  | nil => b         (* Base case: 遇到 nil，替换为 b *)
  | h :: t => f h (fold f t b) (* Recursion: 用 f 替换 cons *)
  end.
```

  - **f : X -> Y -> Y**：用户的结构变换逻辑
  - 本质而言，**fold**在“重写”原有结构的构造子
    - **b** (基础值) 替换了**nil**构造子
    - **f** (结构变换逻辑) 替换了**cons**构造子
    - **l** (参数)是被**fold**的对象结构

规约函数 fold

- 计算和

```
Compute fold plus [1; 2; 3] 0.
(* Coq 输出结果:
   = 6
   : nat
*)
```

- 计算积

```
Compute fold mult [1; 2; 3; 4] 1.
(* Coq 输出结果:
   = 24
   : nat
*)
```

- 计算平均数

```
Require Import Nat. (* 必须引入，它带来了 +, -, *, / 的定义 *)

Definition average_hybrid (l : list nat) : nat :=
  (* Step 1: Fold 计算 (Sum, Count) *)
  let result : prod nat nat :=
    fold
      (fun (x : nat) (acc : prod nat nat) =>
         match acc with
         | pair _ _ s c =>
             (* 这里使用了 + 符号，但保留了 pair 构造子 *)
             pair nat nat (s + x) (c + 1)
         end)
      l
      (pair nat nat 0 0) (* 初始值 *)
in
(* Step 2: 结果计算 *)
match result with
| pair _ _ final_sum final_count =>
    if final_count =? 0
    then 0
    else final_sum / final_count (* 使用除法符号 *)
end.
```

= 逻辑相等

?= 数值相等

规约函数 fold

- 使用fold实现列表的拼接函数app

```
Definition app_fold {X} (l1 l2 : list X) :=
  fold cons l1 l2.
```

一些说明

- 不同的结构化数据类型，对应不同的`filter/map/fold`实现
  - 不同数据类型对应的函数名称不同
    - `list: map | tree: tree_map | option: option_map`
- 不同函数可适用的结构化类型有区别
  - `map`：适用于所有类型
  - `fold`：适用于所有递归的类型
  - `filter`：适用于所有变长/可裁剪的类型

混合使用

- 计算奇数平方和

```
(* 定义辅助函数 *)
Definition is_even (n:nat) := evenb n.
Definition square (n:nat) := n * n.

(* 组合调用：注意阅读顺序是从内向外 *)
Compute fold plus
          (map square
            (filter is_even [1; 2; 3; 4]))
          0.

(* 执行步骤:
   1. inner: filter is_even [1;2;3;4] -> [2; 4]
   2. mid:   map square [2; 4]         -> [4; 16]
   3. outer: fold plus [4; 16] 0       -> 20
*)
(* 结果: 20 *)
```

混合使用

- 计算所有数的平方和

```
(* 1. 定义基础组件 *)
Definition square (n:nat) := n * n.
Definition is_odd (n:nat) := negb (evenb n).

(* 2. 定义通用流水线: 输入一堆数 -> 平方 -> 求和 *)
(* Pipeline: List -> Map -> Fold -> Result    *)
Definition sum_sq_pipeline (l : list nat) : nat :=
  fold plus (map square l) 0.

(* 3. 分组计算 *)

(* 路径 A: 筛选奇数 -> 进流水线 *)
Definition sum_odd_sq (l : list nat) :=
  sum_sq_pipeline (filter is_odd l).

(* 路径 B: 筛选偶数 -> 进流水线 *)
Definition sum_even_sq (l : list nat) :=
  sum_sq_pipeline (filter evenb l).
```

```
(* 定义测试数据 *)
Definition my_list := [1; 2; 3; 4].

(* 方法 1: 分别计算 *)
Compute sum_odd_sq my_list.  (* [1;3] -> 1+9 = 10 *)
Compute sum_even_sq my_list. (* [2;4] -> 4+16 = 20 *)

(* 方法 2: 合并结果 *)
Compute (sum_odd_sq my_list) + (sum_even_sq my_list).
(* 结果: 30 *)

(* 方法 3: 对照组 (直接计算所有) *)
Compute sum_sq_pipeline my_list.
(* 结果: 30 *)
```

Map-reduce 模型

- 分而治之 Divide and Conquer
- Map阶段：完成数据的映射和转换
  - 在并行环境中无需上锁，无需担心数据竞争
    - `filter`与`map`的纯函数特性使得其不会修改外部变量
    - `filter`与`map`只作用于当前元素，不依赖于其他元素
- Reduce阶段：完成数据的规约
  - 并行环境中无需考虑计算顺序和层级
    - `fold`函数满足结合律

Part3- 高阶策略

coq的指令

- Coq如何处理I/O？
  - coq的指令 commands
  - 查询当前的context并输出结果

指令	作用	示例
compute	计算/归约	输入：compute (1 + 1). 输出：= 2 : nat
Check	检查类型	输入：Check true. 输出：bool
Print	打印定义	输入：Print oddb. 输出：显示 oddb 的代码
Search	搜索定理	输入：Search "associative". 输出：列出所有名字里带“结合律”的定理

coq的策略：证明模式的指令

- 函数式程序视角下的证明
  - 从条件context到结果goal的函数调用过程
- 如何保证证明的正确性？
  - 基于规则和公理的推导过程
    - coq支持的
    - 能够自动拼接的
    - 证明函数的调用过程

- Proofs = Types + Functions

  = Tactics(context, goal) -> functions

策略的本质

- 目标：证明 **forall A : Prop, A -> A**
- Coq程序：构造一个函数，输入**A**公式，输出**A**公式成立的证据

```
1  Theorem my_id_tactic : forall A : Prop, A -> A.
2  Proof.
3    intros A x.
4    exact x.
5  Qed.
6
7  Print my_id_tactic.
```

Proof finished

➢ 使用**print** 打印**my_id_tactic**

```
my_id_tactic =
fun (A : Prop) (x : A) => x
     : forall A : Prop, A -> A
```

函数

函数的类型

Curry-Howard Correspondence

```
my_id_tactic =
fun (A : Prop) (x : A) => x
      : forall A : Prop, A -> A
```

coq 代码对应	程序员视角	逻辑学家视角
forall A, A -> A	函数的类型	定理内容
fun(A:Prop)(x:A) => x	函数的实现	证明过程
(x : A)	函数的参数	上下文
x	函数的返回值	推导结论

apply策略

- 逆向推理

- context设定
  - 证明的目标：B
  - 已有条件：规则H: A → B
- apply的直觉解释
  - 需要证明B，同时发现如果证明了A，那么就可以用H推导出B
  - 所以我的目标从证明B变为证明A

apply策略的使用

- apply [定理名称/假设名称]
  - 定理的结论必须和当前的goal完全匹配
  - 当前的goal被消除，更新为所使用定理的前提

```
Parameter Person : Type.
Parameter Man : Person -> Prop.
Parameter Mortal : Person -> Prop.
Parameter Socrates : Person.

Section ApplyDemo.

  Variable H : forall x : Person, Man x -> Mortal x.

  Theorem socrates_will_die : Man Socrates -> Mortal Socrates.
  Proof.
    intros H_man.
    apply H.
    exact H_man.
  Qed.

End ApplyDemo.
```

```
H : forall x : Person, Man x -> Mortal x
H_man : Man Socrates

(1/1)
Mortal Socrates
```

apply策略的使用

- apply [定理名称/假设名称].
  - 定理的结论必须和当前的goal完全匹配
  - 当前的goal被消除，更新为所使用定理的前提

```
Parameter Person : Type.
Parameter Man : Person -> Prop.
Parameter Mortal : Person -> Prop.
Parameter Socrates : Person.

Section ApplyDemo.

  Variable H : forall x : Person, Man x -> Mortal x.

  Theorem socrates_will_die : Man Socrates -> Mortal Socrates.
  Proof.
    intros H_man.
    apply H.
    exact H_man.
  Qed.

End ApplyDemo.
```

```
H : forall x : Person, Man x -> Mortal x
H_man : Man Socrates

(1/1)
Man Socrates
```

apply策略的实现

- coq的证明是在构造一个从前提到结论的函数

- 执行apply H：在证明过程中声明，目标B是通过函数H构造出来的
  - 执行前：有一个需要证明的?Goal_1，类型为B
  - 执行中：确认H的返回类型时B，所以可以使用函数调用H(?Goal_2)替代原来的?Goal_1
  - 执行后：留下了新的需要证明的?Goal_2，类型为A

injection策略

- 拆包装

- context设定
  - 存在一个假设H: con n = con m
    - con是一个结构化数据类型A的构造子
    - con n和con m均为该结构化数据类型的项
- injection的直觉解释
  - 如果两个项con n和con m的类型相同，同时他们的构造子也相同
  - 那么两个项的构造子的参数项m和n也一定相同

injection策略的使用

- **injection [假设名] as [新名字1][新名字2]…**
  - 假设必须是由同一个构造子建立的等式
  - 假设中最外层的构造子**con**会被剥离，其内部参数的等式 **n = m**会作为新的假设添加到上下文中
  - 如果不使用**as**，则等式**n = m**会被添加到**goal**的蕴含前提中**n = m → goal**

```
Theorem injection_demo : forall (n m : nat), S n = S m -> n = m.
Proof.
  intros n m H.
  injection H as H_inner.
  exact H_inner.
Qed.
```

```
n, m : nat
H : S n = S m
(1/1)
n = m
```

injection策略的使用

- injection [假设名] as [新名字1][新名字2]....
  - 假设必须是由同一个构造子建立的等式
  - 假设中最外层的构造子con会被剥离，其内部参数的等式 n = m会作为新的假设添加到上下文中
  - 如果不使用as，则等式n = m会被添加到goal的蕴含前提中n = m → goal

```
Theorem injection_demo : forall (n m : nat), S n = S m -> n = m.
Proof.
  intros n m H.
  injection H as H_inner.
  exact H_inner.
Qed.
```

```
n, m : nat
H_inner : n = m
(1/1)
n = m
```

injection策略的实现

- coq中的构造子是一个单射函数
  - 单设函数满足单射函数定理：

    $\forall\ x\ y,\ f(x) = f(y) \rightarrow x = y$
  - injection本质上就是在使用单射函数定理这个函数

```
Definition injective {A B : Type} (f : A -> B) :=
  forall x y : A, f x = f y -> x = y.
```

discriminate策略

- 矛盾
- context设定
  - 存在一个假设H： con_1 n = con_2 m
    - con_1和con_2是一个结构化数据类型A的不同的构造子
- injection的直觉解释
  - 如果两个项相等，但是他们的构造子不同
  - 那么当前的context存在矛盾
  - 故推理结束

discriminate策略的使用

- discriminate [假设名].
  - 假设必须是一个等式
  - 等式两边最外层的构造子不同
  - 判定前提为假，直接结束证明

```
Require Import Coq.Arith.Arith.

Theorem discriminate_demo : forall (n : nat), 0 = S n -> 1 + 1 = 3.
Proof.
  intros n H.
  discriminate H.

Qed.
```

```
n : nat
H : 0 = S n
(1/1)
1 + 1 = 3
```

discriminate策略的使用

- discriminate [假设名].
  - 假设必须是一个等式
  - 等式两边最外层的构造子不同
  - 判定前提为假，直接结束证明

```
Require Import Coq.Arith.Arith.

Require Import Coq.Arith.Arith.

Theorem discriminate_demo : forall (n : nat), 0 = S n -> 1 + 1 = 3.
Proof.
  intros n H.
  discriminate H.

Qed.
```

```
n : nat

Proof finished
```

discriminate策略的实现

- coq中的构造子需满足值域互斥的定理
  - 对自然数而言，forall n, 0 != S n
  - discriminate则是在应用值域互斥定理

```
Theorem O_neq_S : forall (n : nat), 0 <> S n.
```

- 不同的结构化数据对应的值域互斥定理不同
  - coq实际上采用了一种分情况讨论的证明技巧
  - 当discriminate被调用时：
    - 构造一个函数fun，输入为对应的数据类型
    - match with con_1, 返回值为true
    - match with con_2/其他cons, 返回值为false
    - fun (n) <> fun (m)

destruct策略

- 分类讨论

- context设定
  - 需要证明goal对于一个非递归类型的所有变量均成立
- destruction的直觉解释：
  - case con_1：创建一个subgoal，条件为con_1，目标为当前的goal
  - case con_2：创建一个subgoal，条件为con_2，目标为当前的goal

destruct策略的使用

- **destruct [n] as [n1 | n2 ...].**
  - **n**：需要处理的结构化数据
  - **n1**：**match with** 第一个构造子的变量别名
  - **n2**：**match with** 第二个构造子的变量别名

```
Theorem negb_involutive : forall b : bool, negb (negb b) = b.
Proof.
  intros b.
  destruct b.
  (* --- Case 1: b = true --- *)
    simpl.
    reflexivity.
  (* --- Case 2: b = false --- *)
  -
    simpl.
    reflexivity.
Qed.
```

```
b : bool
(1/1)
negb (negb b) = b
```

destruct策略的使用

- destruct [n] as [n1 | n2 ...].
  - n： 需要处理的结构化数据
  - n1： match with 第一个构造子的变量别名
  - n2： match with 第二个构造子的变量别名

```
Theorem negb_involutive : forall b : bool, negb (negb b) = b.
Proof.
  intros b.
  destruct b.
  (* --- Case 1: b = true --- *)
    simpl.
    reflexivity.
  (* --- Case 2: b = false --- *)
  -
    simpl.
    reflexivity.
Qed.
```

```
(1/2)
negb (negb true) = true
(2/2)
negb (negb false) = false
```

destruct策略的实现

- 基于非结构化数据的定义，构造一个函数
  - 输入：目标goal，con_1的证明序列， con_2的证明序列
  - 输出：能够处理所有该类型变量的证明序列

```
fun (b : bool) =>
  match b with
  | true  => (* 填入 Case 1 的证明: eq_refl *)
  | false => (* 填入 Case 2 的证明: eq_refl *)
  end
```

  - destruct也能够处理递归数据类型，只要goal的证明不涉及递归的过程
    - 证明对于任何nat，S (pred n) = n

induction策略

- 结构归纳

- context设定
  - 需要证明goal对于一个递归类型的变量均成立
    - destruct讨论非递归类型的构造子，无法进行归纳
- induction的直觉解释：
  - case 0：创建一个subgoal，条件为con_b (非递归构造子)，目标为当前的goal
  - case con_1：创建一个subgoal，
    - 条件为归纳假设IH：goal对于m已成立
    - 目标为证明goal对于n = con_1 m成立

induction策略的使用

- induction [n] as [| n' IH].
  - n: 需要处理的结构化数据
  - n': match with 递归构造子的变量别名
  - IH: n的子结构n'的归纳假设 (coq自动生成)

```
Require Import Coq.Arith.Arith.

Theorem plus_n_O : forall n : nat, n + 0 = n.
Proof.
  induction n as [| n' IH].
    reflexivity.
    simpl.
    rewrite IH.
    reflexivity.
Qed.
```

```
MAIN 1    SHELVED 0
(1/1)
forall n : nat, n + 0 = n
```

induction策略的使用

- induction [n] as [| n′ IH].
  - n: 需要处理的结构化数据
  - n′: match with 递归构造子的变量别名
  - IH: n的子结构n′的归纳假设 (coq自动生成)

```
Require Import Coq.Arith.Arith.

Theorem plus_n_O : forall n : nat, n + 0 = n.
Proof.
  induction n as [| n' IH].
    reflexivity.
    simpl.
    rewrite IH.
    reflexivity.
Qed.
```

```
(1/2)
0 + 0 = 0
(2/2)
S n' + 0 = S n'
```

induction策略的使用

- induction [n] as [| n' IH].
  - n: 需要处理的结构化数据
  - n': match with 递归构造子的变量别名
  - IH: n的子结构n'的归纳假设 (coq自动生成)

```
Require Import Coq.Arith.Arith.

Theorem plus_n_O : forall n : nat, n + 0 = n.
Proof.
  induction n as [| n' IH].
    reflexivity.
    simpl.
    rewrite IH.
    reflexivity.
Qed.
```

```
n' : nat
IH : n' + 0 = n'

(1/1)
S (n' + 0) = S n'
```

induction策略的使用

- induction [n] as [| n1 IH1 | n2 IH2…].
  - n存在多个递归构造子时，需要分别处理这些构造子
  - 如果一个构造子涉及了多个子结构，则需要引入多条IH

```
Inductive Tree3 : Type :=
  | Leaf : Tree3                      (* 构造子 1: 根节点 *)
  | Node1 : Tree3 -> Tree3            (* 构造子 2: I型节点 *)
  | Node2 : Tree3 -> Tree3 -> Tree3.  (* 构造子 3: Y型节点 *)

Theorem tree_induction_demo : forall t : Tree3, t = t.
Proof.
  induction t as [ | t1 IH1 | t2a IH2a t2b IH2b ].
  (* --- 分支 1: Leaf (Base Case) --- *)
  - reflexivity.
  (* --- 分支 2: Node1 (Step Case 1) --- *)
  - simpl. reflexivity.
  (* --- 分支 3: Node2 (Step Case 2) --- *)
  - simpl. reflexivity.
Qed.
```

```
(1/3)
Leaf = Leaf

(2/3)
Node1 t1 = Node1 t1

(3/3)
Node2 t2a t2b = Node2 t2a t2b
```

induction策略的实现

- 基于结构化数据的结构归纳定义，构造对应的递归函数
  - 基于induction策略的证明：编写递归函数
  - 归纳假设IH：递归函数调用的返回值
- coq为自然数nat自动生成了“通用调用模板”nat_ind
  - **输入：** 目标goal，非递归构造子(n = 0)的证明序列，递归构造子(n = S n')的证明序列
  - **输出：** 能够处理所有自然数的证明序列

```
Print plus_n_O.
(* 核心结构如下 (简化版): *)
(* nat_ind (fun n => n + 0 = n)
           (Base_Case_Proof)
           (fun n IH => ... Step_Case_Proof ...)
*)
```