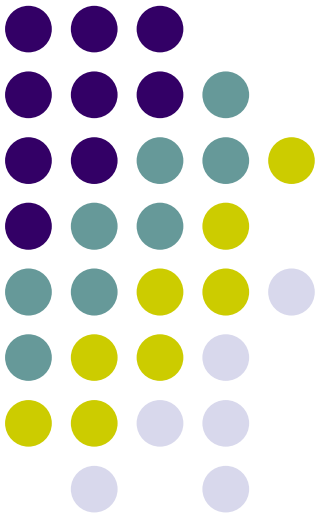
南京大学
Nanjing University

第十讲- coq中的数学计算、逻辑构造与推理证明



内容提要

- 数学计算
  - 从自然数到整数 | 命题
  - 线性整数算术 lia
- 逻辑构造
  - 逻辑连接词及其coq实现
  - 公式与替换
- 推理证明
  - 公式的证明策略
  - 自定义推理规则
  - 自动化策略

Part1- 数学计算

自然数nat的局限性

- nat是完美的归纳结构，但是：
  - 对于四则运算不封闭
    - 3 - 5 = -2（数学上）
    - 3 - 5 = 0 （基于nat的定义）
  - 存储方式效率低下
    - nat本质上是一种一进制的计数方法
    - 10 = S (S (S (S (S (S (S (S (S (S 0))))))))))
  - 计算过程非常繁琐
    - 10 + 10 涉及20层的结构拆解+20层的结构规约

Z：二进制整数

- 由**ZArith**库定义，类似于高级编程语言中的integer类型
- 使用方式：
  - 声明引入**Zarith**库：**Require Import ZArith.**
  - 声明运算作用域为整数：**Open Scope Z_scope.**

```
Require Import ZArith.

(* 场景 A: 在 nat 作用域下 *)
Module Nat_Demo.
  Open Scope nat_scope.
  Compute (3 - 5).
  Lemma bad_math : 3 - 5 <> 0.
  Proof. simpl. Abort.
End Nat_Demo.

(* 场景 B: 在 Z 作用域下 *)
Module Z_Demo.
  Open Scope Z_scope.
  Compute (3 - 5).
  Lemma good_math : 3 - 5 = -2.
  Proof. reflexivity. Qed.
End Z_Demo.
```

```
(1/1)
0 <> 0
```

nat域中3-5=0

Z：二进制整数

- 由**ZArith**库定义，类似于高级编程语言中的integer类型
- 使用方式：
  - 声明引入**ZArith**库：**Require Import ZArith.**
  - 声明运算作用域为整数：**Open Scope Z_scope.**

```
Require Import ZArith.

(* 场景 A: 在 nat 作用域下 *)
Module Nat_Demo.
  Open Scope nat_scope.
  Compute (3 - 5).
  Lemma bad_math : 3 - 5 <> 0.
  Proof. simpl. Abort.
End Nat_Demo.

(* 场景 B: 在 Z 作用域下 *)
Module Z_Demo.
  Open Scope Z_scope.
  Compute (3 - 5).
  Lemma good_math : 3 - 5 = -2.
  Proof. simpl. reflexivity. Qed.
End Z_Demo.
```

```
(1/1)
-2 = -2
```

Z域中
3-5=-2

Z的实现

- 用归纳结构模拟二进制代码
  - 正整数：**5 = 101（二进制）= xI (x0 xH)**

```
Inductive positive : Set :=
  | xH : positive            (* 1 (二进制 1) *)
  | xO : positive -> positive  (* p => 2 * p (在二进制末尾补 0) *)
  | xI : positive -> positive. (* p => 2 * p + 1 (在二进制末尾补 1) *)
```

  - 整数：**Zpos**和**Zneg**构造子分别处理正整数和负整数
    - **coq**并不知道**Zpos**和**Zneg**的含义，它只知道两者在参与计算时会被不同的函数（计算规则）匹配 **(match with)**

```
Inductive Z : Set :=
  | Z0 : Z                   (* 0 *)
  | Zpos : positive -> Z     (* +p *)
  | Zneg : positive -> Z.    (* -p *)
```

Z的实现

- 用归纳结构模拟二进制代码
  - 整数：**Zpos**和**Zneg**构造子分别处理正整数和负整数
    - **coq**并不知道**Zpos**和**Zneg**的含义，它只知道两者在参与计算时会被不同的函数（计算规则）匹配 **(match with)**

```
Definition Z_add (x y : Z) : Z :=
  match x, y with
  | Z0, _ => y
  | _, Z0 => x
  (* 情况 A: 同号相加 -> 数值相加，符号保留 *)
  | Zpos p, Zpos q => Zpos (p + q)
  | Zneg p, Zneg q => Zneg (p + q)
  (* 情况 B: 异号相加 -> 数值相减，符号取决于谁绝对值大 *)
  | Zpos p, Zneg q => pos_sub p q  (* 正 + 负 = 减法 *)
  | Zneg p, Zpos q => pos_sub q p  (* 负 + 正 = 减法 *)
  end.
```

浮点数与实数

- **Floats**库定义用于计算的浮点数 **Float**
  - 直接实现符合**IEEE 754 Binary64**标准的浮点数
  - 直接调用**IEEE 754**的指令集，支撑高效的计算过程
  - **面向计算需求，但是不能用于证明！**
- **Reals**库定义了用于证明的实数 **R**
  - 基于实数公理定义，无法给出一个具体的值
  - **用于证明各类型定理，但不支持计算！**
- **Flocq**库提供了结构化定义的浮点数
  - 类似于**Z**，基于整数模拟浮点数
  - 建立了**R**和**Float**的二元关系，**支持用Float逼近任意R！**

浮点数与实数：Float

```
Require Import Coq.Floats.Floats.
Open Scope float_scope.
(* --- 演示 A: 快速计算 --- *)
Definition float_calc := sqrt 2 + 1.
Compute float_calc.
(* --- 演示 B: 精度丢失（舍入误差）--- *)
Definition one_third := 1 / 3.
Definition sum_three := one_third * 3.

Compute (sum_three = 1).
```

```
:MS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORT
= 2.4142135623730949
: float
```

```
Require Import Coq.Floats.Floats.
Open Scope float_scope.
(* --- 演示 A: 快速计算 --- *)
Definition float_calc := sqrt 2 + 1.
Compute float_calc.
(* --- 演示 B: 精度丢失（舍入误差）--- *)
Definition p1 := 0.1 + 0.2.
Definition p2 := 0.3.
Compute (p1 = p2).
Compute (p1 - p2).
```

```
:MS 3    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS
= 0.30000000000000004 = 0.29999999999999999
: Prop
= 5.5511151231257827e-017
: float
```

浮点数与实数：Real

```
1   Require Import Coq.Reals.Reals.
2   Open Scope R_scope.
3
4   Compute (sqrt 2).
5
6   Theorem real_sqrt_exact : sqrt 2 * sqrt 2 = 2.
7   Proof.
8     apply sqrt_sqrt.
9     apply Rlt_le.
10    apply Rlt_0_2.
11  Qed.
12
```

ROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Filter (e.g. text, !excludeText, t...

```
  := match Rcase_abs (R1 + R1) with
    | left _ => R0
    | right x =>
        let (x0, _) := Rsqrt_exists (R1 + R1) (Rge_le (R1 + R1) R0 x) in
        x0
    end
  : R
```

对实数2 (R1+R1)进行分情况讨论：

1. left(负数)，则返回值为R0
2. right(正数)，则返回值为平方根公理所规定的平方根x0

x0是实数，故sqrt 2是一个实数

命题 Prop

- 考虑 `x < 10`
  - 朴素语义：这是一个可以被判断真假的句子
  - 过程式程序设计语言：这是一个返回值为`bool`的表达式
  - `coq`：这是一个特殊的类型，称为`Prop`

- `Prop`
  - 值是这个命题的证明 `proof`
  - 使用命题时，需要进行模型匹配`match with`
- `Prop`可以表达不能进行朴素“计算”的命题
  - `forall X, exists Y, Y > X`

命题 Prop

```
Require Import ZArith.
Open Scope Z_scope.

Compute (2 < 3).

Compute (2 <? 3).
```

```
:MS    OUTPUT    DEBUG CONSOLE
= Lt = Lt
: Prop
```

```
Require Import ZArith.
Open Scope Z_scope.

Compute (2 < 3).

Compute (2 <? 3).
```

```
BLEMS    OUTPUT    DEBUG CONSOL
= Lt = Lt
: Prop
= true
: bool
```

复杂的命题

- 命题可以用于Z域中的复杂约束
  - 寻找一个整数x，使得
    - 在区间[0, 10]内
    - 且是偶数
    - 且不等于5

```
Definition valid_number (x : Z) : Prop :=
  0 <= x <= 10 /\       (* 区间约束 (语法糖: 0 <= x /\ x <= 10) *)
  (exists k, x = 2 * k) /\ (* 偶数约束 *)
  x <> 5.               (* 排除约束 *)
```

/\	合取	->	蕴含	forall	全称
\/	析取	~	否定	exists	存在

线性整数算术 lia

- **Linear Integer Arithmetic solver**
  - 能够自动判定整数相关的线性命题是否可证
    - 不能处理非线性问题（变量相乘，取模，指数等）

```
Require Import ZArith.
Require Import Lia.
Open Scope Z_scope.

Theorem interval_squeeze : forall x y : Z,
  0 < x ->              (* 前提 1: x 是正数 *)
  x < y ->              (* 前提 2: x 小于 y *)
  y < x + 2 ->          (* 前提 3: y 小于 x + 2 *)
  y = x + 1.            (* 结论: y 只能是 x + 1 *)
Proof.
  intros.
  lia.
Qed.
```

Proof finished

线性整数算术 lia

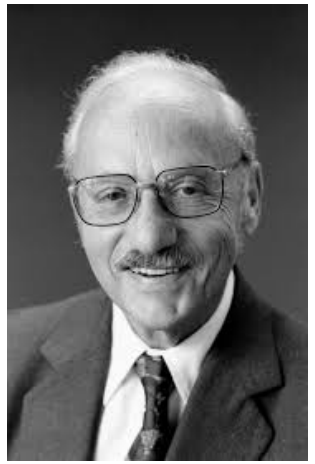
```
Open Scope Z_scope.

Theorem interval_squeeze : forall x y : Z,
  0 < x ->              (* 前提 1: x 是正数 *)
  x < y ->              (* 前提 2: x 小于 y *)
  y < x + 2 ->          (* 前提 3: y 小于 x + 2 *)
  y = x + 1.            (* 结论: y 只能是 x + 1 *)
Proof.
  intros.
  lia.
Qed.

Theorem no_integer_solution : forall x y : Z,
  2 * x + 4 * y <> 1. (* 结论: 不等于 1 *)
Proof.
  intros.
  intro H_contra.
  lia.
Qed.
```

```
MAIN 1   SHELVED 0   GIVE

x, y : Z
H_contra : 2 * x + 4 * y = 1

(1/1)
False
```

将前提**2*x+4*y=1**剥离为一个假设

lia不仅能够证明逻辑上的肯定，
还可以证明逻辑上的否定

lia的实现机制*



- 单纯形法 + 割平面法

- 单纯形法：Simplex Method
  - 能够处理实数域上的线性规划问题
  - 在整数域上面临整数间隙问题 1 < x < 3

- 割平面法：从解空间中去掉单纯形法的解
  - 如果单纯形法返回了1.5，则去掉包含1.5且不包含任何整数的解空间子集
  - **NP问题！难度远大于线性规划**

Part2- 逻辑构造

计算视角与逻辑视角

- `2 <? 3 vs. 2 < 3`

- 一个coq表达式到底表示一个**项**，还是表示一个**命题**
  - **项**：计算视角下的表达式，用于参与计算
    - `3, [1;2], x+y, 2 <? 3, true`
  - **命题**：逻辑视角下的表达式，用于参与推理
    - `3=3, x<y, 2<3, True`
  - 一个表达式不可能同时是项和命题
    - coq的计算域和逻辑域是严格分离的

可执行程序与逻辑证明

- coq即是程序设计语言，又是证明器
  - 程序设计语言：计算域
    - 参与底层运算
    - 通过将coq代码转化为Haskell代码实现

```
Compute (2 <? 3).
(* Result: true *)
(* 结论：这是一个计算过程，机器直接给出了答案。 *)
```

  - 证明器：逻辑域
    - 用于保障底层运算的正确性
    - 通过编译过程的类型检查机制实现

```
Compute (2 < 3).
(* Result: (2 ?= 3) = Lt *)
(* 结论：这不是答案，这只是把问题换了一种说法（展开定义）。 *)
```

coq的逻辑系统

- 和Z，Float一样，coq的逻辑系统也是模拟的！

- coq的内核并不认识大部分逻辑连接词和量词
  - /\和+没有本质区别
  - &&和List都是某种类型
- coq的逻辑就是对各种结构和函数进行类型检查
  - 结构：Prop，nat，Z
  - 函数：A -> B -> C

Proposition is type, proof is program!

coq的逻辑原语*

- 类型检查需要做什么？
  - 检查结构化数据就是在检查函数(构造子)
    - 检查`nat`，只需要检查1) 常函数 `O`; 2) `nat -> nat` 类型函数 `S n`
    - 检查`list`，只需要检查1) 常函数 `nil`; 2) `X-> list-> list`类型函数 `X-> list-> list`
  - 检查函数需要
    - 检查上下文：函数的参数是什么，参数的类型是什么
    - 检查依赖性：返回值依赖于参数

      需要有一个原语连接（参数 ： 类型）和 返回值

coq的逻辑原语*

（参数：类型） 返回值就是**forall （x: A), B**

- 检查上下文：声明了**x**是类型为**A**的参数
- 检查依赖性：输出**B**是一个关于**x**的命题**B(x)**
  - 如果**B**和输入参数**x**个体无关，但是和其类型**A**相关，则该函数就是蕴含
- 在**coq**的内核**CIC**中，**forall**被定义为依赖积**Πx:A,B(x)**
  - 接收类型为**A**的参数**x**，返回类型为**B(x)**的值

coq的True

- **True**：用于表征“正确”的类型
  - 被定义为一个单例归纳类型

```
Inductive True : Prop :=
  | I : True.  (* 唯一的构造子，名字叫 I *)
```

  - 要证明**True**只需要给出构造子**I**即可

```
Lemma prove_true : True.
Proof.
  exact I.  (* 直接提供构造子 *)
  (* 或者使用策略: split. *)
Qed.
```

coq的False

- **False**：用于表征“矛盾”的类型
  - 被定义为一个空的归纳类型

```
Inductive False : Prop := .
(* 注意：这里什么都没有！ *)
```

  - 在**coq**中通常不可能证明**False**
  - 当**False**出现在上下文中时，可以证明任何内容

```
Lemma ex_falso_quodlibet : False -> 1 = 2.
Proof.
  intros H.       (* 假设我们拿到了一个 False 的证据 H *)
  destruct H.     (* 试图拆解 H。因为 False 没有构造子，
                     Coq 发现情况数为 0，直接判定证明完成。 *)
Qed.
```

coq的True和False

- 证明True/False与证明一个公式为True/False完全不同！
  - 证明True/False是在证明context可以被规约到I或者空
  - 证明公式为True/False则依赖于公式具体的形式

```
Lemma prove_eq : 1 + 1 = 2.
Proof.
  reflexivity.  (* 战术：去计算并检查两边是否一样 *)
Qed.

Print prove_eq.
(* 输出: eq_refl 2 *)
```

证明等式需要证据eq_refl

coq的True和False

- 证明True/False与证明一个公式为True/False完全不同！
  - 证明True/False是在证明context可以被规约到I或者空
  - 证明公式为True/False则依赖于公式具体的形式

```
Lemma prove_imp : 1 = 1 -> 2 = 2.
Proof.
  intro H.       (* 战术：接收输入参数，命名为 H *)
  reflexivity.   (* 战术：构造输出值 *)
Qed.
```

证明蕴含需要构造函数fun

```
Print prove_imp.
(* 输出: fun (H : 1 = 1) => eq_refl 2 *)
```

coq的not

- **not A**：函数 **A -> False**，
  - 如果参数**A**成立，则能够推导出**False**

```
Definition not (A : Prop) := A -> False.
```

  - 输入**A**， 输出**False**

- 证明**not A**，就是在证明函数**A -> False**成立

```
Lemma prove_not_eq : ~ (1 = 2).
Proof.
  intro H.          (* 假设 1=2 *)
  discriminate H. (* 1和2的构造不同，所以H是矛盾的*)
Qed.
```

```
H :  1 = 2

(1/1)
False
```

coq的and

- **A /\ B**：构造一个非递归结构，由**A**和**B**组成一个**pair**
  - 接收**A**和**B**两个参数，类型为**and A B**

```
Inductive and (A B : Prop) : Prop :=
  | conj : A -> B -> and A B.
```

- 当**A /\ B**作为**goal**时，使用**split**将其拆分为**A**和**B**两个子**goal**

```
Lemma prove_and : 1=1 /\ 2=2.
Proof.
  split.        (* 目标分裂为两个 *)
  - reflexivity. (* 证明 1=1 *)
  - reflexivity. (* 证明 2=2 *)
Qed.
```

```
(1/2)
1 = 1

(2/2)
2 = 2
```

coq的and

- **A /\ B**：构造一个非递归结构，由**A**和**B**组成一个**pair**
  - 接收**A**和**B**两个参数，类型为**and A B**

```
Inductive and (A B : Prop) : Prop :=
  | conj : A -> B -> and A B.
```

- 当**A /\ B**作为**context**时，使用**destruct**将其拆分为**HA:A**和**HB:B**两个**context**

```
Lemma use_and : forall A B : Prop, A /\ B -> A.
Proof.
  intros A B H.
  destruct H as [HA HB]. (* 拆包：HA是A的证据，HB是B的证据 *)
  exact HA.              (* 我们需要A，正好手里有HA *)
Qed.
```

```
MAIN
A, B : Prop
HA : A
HB : B
(1/1)
A
```

coq的or

- **A \/ B**：构造一个非递归结构，有两个构造子分别对应**A**和**B**
  - 构造子接收一个**A(或者B)**参数，类型为**or A B**

```
Inductive or (A B : Prop) : Prop :=
  | or_introl : A -> or A B  (* 左路构造子：只要有 A，就能造出 (A \/ B) *)
  | or_intror : B -> or A B. (* 右路构造子：只要有 B，就能造出 (A \/ B) *)
```

- 当**A \/ B**作为**goal**时，需要选择**or_introl**或**or_intror**构造子拆解**A \/ B**

```
Theorem go_left : forall (A B : Prop), A -> A \\// B.
Proof.
  intros A B HA.
  (* 目标：A \\// B *)
  (* 策略：因为我有 A，所以我选择走左边 (Left) *)
  apply or_introl.
  exact HA.
Qed.
```

```
A, B : Prop
HA : A
(1/1)
A
```

coq的or

- **A \/ B**：构造一个非递归结构，有两个构造子分别对应**A**和**B**
  - 构造子接收一个**A(或者B)**参数，类型为**or A B**

```
Inductive or (A B : Prop) : Prop :=
  | or_introl : A -> or A B  (* 左路构造子：只要有 A，就能造出 (A \/ B) *)
  | or_intror : B -> or A B. (* 右路构造子：只要有 B，就能造出 (A \/ B) *)
```

- 当**A \/ B**作为**context**时，需要使用**destruct**同时处理两个构造子

```
(* 交换律证明：如果 A \/ B，那么 B \/ A *)
Theorem or_comm : forall (A B : Prop), A \\// B -> B \\// A.
Proof.
  intros A B H_choice.
  destruct H_choice as [HA | HB].
  (* 情况 1: H_choice 里面装的是 A *)
    apply or_intror. (* 走右路 *)
    exact HA.
  (* 情况 2: H_choice 里面装的是 B *)
    apply or_introl. (* 走左路 *)
    exact HB.
Qed.
```

```
A, B : Prop
HA : A

(1/2)
B \\// A

(2/2)
B \\// A
```

coq的imply

- A -> B由forall原语直接支持

- 当A -> B作为goal时，需要构造对应的A->B函数
  - 在HA: A的context下，证明新的goal B

```
(* 1. 证明蕴含：就是写一个匿名函数 (Lambda) *)
(* 目标：证明 "如果 A 成立，那么 A 就成立" (Identity Function) *)
Theorem my_id : forall A : Prop, A -> A.
Proof.
  intros A.
  intro H_proof.
  exact H_proof.
Qed.
```

```
A :  Prop
H_proof :  A

(1/1)
A
```

coq的imply

- A -> B由forall原语直接支持

- 当A -> B作为context时，需要调用函数A -> B
  - 在A -> B的context下，证明新的goal A

```
21  (* 2. 使用蕴含：就是函数调用 (Modus Ponens) *)
22  (* 目标：如果 (A->B) 且 (A成立)，那么 (B成立) *)
23  Theorem modus_ponens : forall A B : Prop, (A -> B) -> A -> B.
24  Proof.
25    intros A B.
26    intros H_func. (* H_func 是一个函数:  A -> B *)
27    intros H_arg.  (* H_arg  是一个参数:  A *)
28    apply H_func.
29    exact H_arg.   (* 我们手里正好有 H_arg : A *)
30  Qed.
```

```
A, B :  Prop
H_func :  A -> B
(1/1)
A -> B
```

coq的exists

- **exists x, P x**：构造一个非递归结构，由个体**x**和**x**满足性质**P**的证据**P x**组成
  - 元素**x**和**P x**是依赖关系

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  | ex_intro : forall x : A, P x -> ex A P.
```

- 当**exists x, P x**作为目标时，提供可以满足**P**的实际变量值**v**

```
Module MyLogic_Exists.
  Require Import Arith.
  Theorem exists_greater_than_zero : exists n : nat, n > 0.
  Proof.
    exists 1.
    (* Coq 会把目标里的 n 替换成 1。
       现在的目标变成了: 1 > 0 *)
    (* 使用自然数库里自带的定理完成证明 *)
    constructor.
  Qed.
```

```
(1/1)
1 > 0
```

coq的exists

- **exists x, P x**：构造一个非递归结构，由个体x和x满足性质P的证据P x组成
  - 元素**x**和**P x**是依赖关系
- 当**exists x, P x**作为**context**时，使用**destruct**拆开得到元素**x**和**x**满足**P**的**context, P x**

```
(* 证明：如果存在一个数字同时满足 P 和 Q，那么必然存在一个数字满足 P *)
Theorem extract_witness :
  forall (A : Type) (P Q : A -> Prop),
  (exists x : A, P x /\ Q x) -> (exists y : A, P y).
Proof.
  intros A P Q H_exist.
  (* 拆包动作!
     将未知个体命名为 'witness'
     将它的属性证明命名为 'H_and' *)
  destruct H_exist as [witness H_and].
  exists witness.
  destruct H_and as [H_P H_Q].
  exact H_P.
Qed.
```

```
A : Type
P, Q : A -> Prop
witness : A
H_and : P witness /\ Q witness
(1/1)
exists y : A, P y
```

coq中的命题

coq的命题不是一阶语言/命题逻辑的命题！

- 命题 **Prop**：没有任何未知参数的公式
  - 拥有专属的顶级类型**Prop**
  - 可以通过**Definition**或者**Theorem**声明
  - 是一个静态且封闭的，可以进行证明的**coq对象**

```
(* 声明三个基本的命题变量  *)
Variables A B C : Prop.
(* 1. 组合出来的依然是命题 *)
Definition my_first_prop : Prop :=
  (A \/ B) -> C.
(* 2. 包含量词的算术论断也是命题 *)
Definition my_math_prop : Prop :=
  forall x : nat, x >= 0.
```

coq中的谓词

coq的谓词不是一阶语言的命题！

- 谓词：包含自由变元的公式
  - 是一个输入参数为自由变元，返回值类型为**Prop**的函数
  - 需要具体的参数（通过替换或量词约束）才能进行证明
  - 谓词 **x A: Prop**的类型是 **A -> Prop**
    - 一阶语言的谓词和包含自由变元的公式都是**coq的谓词**

```
Require Import Arith.
(* 1. 定义一个谓词：判断一个自然数是否等于 0 *)
(* 它是一个函数，接收 nat，返回 Prop *)
Definition is_zero (x : nat) : Prop :=
  x = 0.
(* 2. 正确用法 A：实例化（传参）-> 变成命题 *)
Definition prop_1 : Prop := is_zero 0.  (* 传入 0 *)
Definition prop_2 : Prop := is_zero 5.  (* 传入 5，这也是命题，只是它是假的 *)
(* 2. 正确用法 B：量化（封闭）-> 变成命题 *)
Definition prop_3 : Prop := forall x : nat, is_zero x.
Definition prop_4 : Prop := exists x : nat, is_zero x.
```

coq的替换

- `P t`: 对于谓词`P`中的自由变元进行`[t/x]`替换
  - coq的谓词通过参数显示化地声明了自由变元
    - `Definition A (x : U) := P x \/ Q y.`中，`x`是自由变元，`y`不是自由变元
  - 替换只会处理自由变元（函数的入口只有自由变元）
  - 如果存在多个自由变元，则需要基于顺序进行替换
    - `Definition B (x y : U) : Prop := P x \/ R x y.`

```
(* 案例 1: 只替换第一个变量 (对应 x) *)
Theorem partial_subst : B t1 y -> (P t1 \/ R t1 y).
Proof.
intro H.
unfold B in H.
exact H.
Qed.

(* 案例 2: 按顺序替换两个变量 *)
```

```
U :  Type
P :  U -> Prop
R :  U -> U -> Prop
t1, t2, y :  U
H :  P t1 \/ R t1 y
(1/1)
P t1 \/ R t1 y
```

coq的替换

- `P t`: 对于谓词`P`中的自由变元进行`[t/x]`替换
  - coq的谓词通过参数显示化地声明了自由变元
    - `Definition A (x : U) := P x \/ Q y.`中，`x`是自由变元，`y`不是自由变元
  - 替换只会处理自由变元（函数的入口只有自由变元）
  - 如果存在多个自由变元，则需要基于顺序进行替换
    - `Definition B (x y : U) : Prop := P x \/ R x y.`

```
(* 案例 2: 按顺序替换两个变量 *)
Theorem full_subst : B t1 t2 -> (P t1 \/ R t1 t2).
Proof.
intro H.
unfold B in H.
exact H.
Qed.

End MyLogic_Targeting.
```

```
U :  Type
P :  U -> Prop
R :  U -> U -> Prop
t1, t2, y :  U
H :  P t1 \/ R t1 t2

(1/1)
P t1 \/ R t1 t2
```

coq视角下的公式

- 带有自由变元的公式（谓词）是函数，将自由变元替换为具体项后的公式（命题）是结构化数据
  - 一阶语言公式在**coq**中是一棵抽象语法树**(AST)**
    - 叶节点：对应于原子谓词的函数 **P x**
    - 中间数据节点：对应于**and, or, exists**的结构化数据
    - 中间函数节点：对应于**not, forall, imply**的函数

coq视角下的公式证明

- 命题的证明过程就是从叶节点出发，重新构造整棵AST
  - 遇到叶节点`(P x)`：使用`context`进行替换
  - 遇到中间函数节点 `(imply, forall, not)`:使用`intro`接收参数
  - 遇到中间数据节点 `(and, or, exists)`:使用 `split, left/right, exists` 构造，或 `destruct` 拆解。

Part3- 推理证明

前向推理

- 前向推理 **Forward reasoning**
  - 从已知条件出发，一步步推导，直到到达结论。
  - **H系统**证明序列的构造过程
  - **coq**中的**assert, post, specialize**策略等

```
(* 2. 前向推理：从前提 A 推导到 B，再到 C *)
Theorem forward_demo : A -> C.
Proof.
  intro HA.
  (* 使用 assert 断言一个中间结论 B 成立 *)
  assert (HB : B). (* 在这个大括号里，我们要专门证明 B *)
  {
    apply H1.
    exact HA.
  }
  (* 现在 HB : B 已经作为一个新的前提加入了环境 *)
  apply H2.
  exact HB.
Qed.
```

```
A, B, C :  Prop
H1 :  A -> B
H2 :  B -> C
HA :  A
HB :  B
──────────────
(1/1)
C
```

后向推理

- 后向推理 **Backward reasoning**
  - 从目标出发，对待证目标进行简化
  - **G系统**证明树的构造过程
  - **coq**中的**apply, intro, split**策略等

```
Section MyLogic_ReasoningDir.

  Variables A B C : Prop.
  Hypothesis H1 : A -> B.
  Hypothesis H2 : B -> C.

  (* 1. 后向推理 (推荐): 从目标 C 倒推到 A *)
  Theorem backward_demo : A -> C.
  Proof.
    intro HA.        (* 动作: 将 A 移入前提 *)
    apply H2.        (* 动作: 为了证 C, 利用 B->C, 现在只需证 B *)
    apply H1.        (* 动作: 为了证 B, 利用 A->B, 现在只需证 A *)
    exact HA.        (* 动作: A 正好在前提里, 证毕 *)
  Qed.
```

```
A, B, C : Prop
H1 : A -> B
H2 : B -> C
HA : A

(1/1)
C
```

像G系统一样使用coq策略

- G系统与coq的对应
  - 序贯Γ ⊢ Δ对应于coq的context -> goal

符号	在 Goal 时	在 Context 时
->	intro 假设前件成立，证明后件	apply 逆向归约
forall	intro 给变量起个名字	apply / specialize 代入具体数值
/\	split 拆成两个子目标	destruct … as [H1 H2] 拆成两个已知条件H1, H2
\/	left 或 right 选一边证明	destruct … as [H1 H2] 拆成两个目标H1, H2
exits	exists x 拿出具体的见证者	destruct … as [x H_proof] 拆出个体x和证据H_proof
not	intro 引入假设“A为false”	apply / destruct 推出矛盾

从人工证明到自动化证明

- coq提供了Ltac原语，支持用户的自定义策略
  - 策略：用于处理context或goal中的特定结构化数据的封装的函数

```
Ltac easy_solve := intros; simpl; reflexivity.
```

- 封装的函数：将若干个原生策略打包为一个策略

```
(* 1. 痛点：重复的机械操作 *)
Theorem boring_proof : (forall x, P x -> P x) /\ (forall y, P y -> P y).
Proof.
  split.
  - intros x H. apply H. (* 重复操作 1 *)
  - intros y H. apply H. (* 重复操作 2 *)
Qed.

(* 2. 解决方案：定义一个 Ltac 宏 *)
```

```
P :  nat -> Prop
H_refl :
  forall x : nat, P x -> P x
(1/2)
forall x : nat, P x -> P x
(2/2)
forall y : nat, P y -> P y
```

从人工证明到自动化证明

- coq提供了Ltac原语，支持用户的自定义策略
  - 策略：用于处理context或goal中的特定结构化数据的封装的函数

```
Ltac easy_solve := intros; simpl; reflexivity.
```

- 封装的函数：将若干个原生策略打包为一个策略

```
(* 2. 解决方案：定义一个 Ltac 宏 *)
Ltac my_solver :=
  intros x H;  (* 引入假设 *)
  apply H.     (* 应用假设 *)

(* 3. 使用自定义策略 *)
Theorem smart_proof : (forall x, P x -> P x) /\ (forall y, P y -> P y).
Proof.
  split.
  - my_solver. (* 自动化调用 *)
  - my_solver. (* 自动化调用 *)
Qed.
```

```
P :  nat -> Prop
H_refl :
  forall x : nat, P x -> P x

(1/2)
forall x : nat, P x -> P x

(2/2)
forall y : nat, P y -> P y
```

应对特定数据

- **Ltac**可以通过模式匹配，基于当前的**goal**和**context**状态，选择不同的证明方法

```
Variables A B C : Prop.
(* 定义一个智能策略：处理各种逻辑连接词 *)
Ltac smart_logic :=
  repeat match goal with
  (* 规则 1: 如果目标是 A /\ B，就拆分 *)
  | [ |- _ /\ _ ] =>
      idtac "Splitting AND..."; split
  (* 规则 2: 如果目标是 A -> B，就引入 *)
  | [ |- _ -> _ ] =>
      idtac "Introducing..."; intro
  (* 规则 3: 如果前提里有 A /\ B，就拆解它 *)
  | [ H : _ /\ _ |- _ ] =>
      idtac "Destructing Hypothesis..."; destruct H
  (* 规则 4: 尝试直接匹配假设 *)
  | [ H : ?P |- ?P ] =>
      exact H
  end.
```

- **|- ：context**和**goal**的分界线
- **_ ：** 通配符，表示当前策略与这个位置上的内容无关
- **? P:** 占位符，表示匹配任意命题并赋值给**P**
- **idatc “…” ：** 在控制台打印字符串内容

coq内置的自动化策略

- 命题逻辑 `tauto`
  - 基于命题逻辑的完全判定算法，可以处理任何只包含∧, ∨, →, ¬ 的纯命题逻辑公式
- 线性整数 `lia`
  - 基于整数线性规划，可以处理包含 +, -, =, <, > 的整数不等式组
- 引理库搜索 `auto / eauto`
  - 利用DFS搜索算法，可以递归地尝试使用已知的引理证明目标

自动化证明的边界

- 理论的天花板
  - **哥德尔不完备定理**
  - **停机问题**

- 工程实现的边界
  - 状态空间爆炸
    - 当 `auto` 尝试寻找证明时，如果在某一步有 10 个引理可以使用，它就要分出 10 个树杈。如果证明需要 20 步，搜索空间就是 `0^20`
    - 默认的`auto`搜索层数是**5层**