定理证明与程序验证

曹钦翔

如果说编程是通过编写代码，将简单的函数、功能组合起来实现复杂的功能，那么证明就是将简单的证明步骤组合起来构成复杂性质的证明。以 Coq、Isabelle 等为代表的定理证明工具让我们用代码的形式来描述数学定义与数学证明。最新版本的 Coq 可以在以下网址下载：

`https://github.com/coq/platform/releases/latest`

本讲义的代码也可以在以下网址下载：

`https://bitbucket.org/qinxiang-SJTU/noiwc2023/src/master/`

1 等式的证明

```
Module Group.
```

这是一个关于群论的证明。首先，我们要定义一个群包含哪些运算。

```
Class GroupOperator: Type := {
  carrier_set: Type;
  zero: carrier_set;
  add: carrier_set -> carrier_set -> carrier_set;
  neg: carrier_set -> carrier_set
}.
```

这里我们可以忽略`Class`、`Type`这些 Coq 保留字，上面这个定义大致说的是：要定义群运算就先要定义一个集合（`carrier_set`），之后一个群应当包含一个单位元（`zero`）、一个二元运算（`add`）以及一个逆元预算（`neg`）。

```
Notation "0" := (zero).
Notation "x + y" := (add x y).
Notation "- x" := (neg x).
```

在 Coq 中，我们可以用零、加号以及负号这些 Notation 来帮助我们表述相关性质。例如：下面就是一个合法的关于群论的命题：

```
Check forall (G: GroupOperator) (x y: carrier_set), x + y = y + x.
```

这里 Coq 的 Check 指令可以理解为检查一个表述语法上是否合法。上面检查的结果是：这是一个合法的 Coq 命题。注意，这只是语法检查，不是证明。下面是群论中的经典证明：从左单位元、左逆元两条性质推出右逆元性质。首先定义：一个群应当具有左单位元、左逆元与结合律这三条性质。

```
Class GroupProperties (G: GroupOperator): Prop := {
  assoc: forall (x y z: carrier_set), (x + y) + z = x + (y + z);
  left_unit: forall (x: carrier_set), 0 + x = x;
  left_inv: forall (x: carrier_set), add (neg x) x = zero
}.
```

其次证明：有上面性质可以推出右逆元性质。

```
Theorem right_inv {G: GroupOperator} {GP: GroupProperties G}:
  (forall (x: carrier_set), x + (- x) = 0).
Proof.
```

在 CoqIDE 中，你可以用 Ctrl+ 向下快捷键让 Coq 检验你的定义与证明。通过检验的代码会变成绿色。进入证明模式后，你会在 CoqIDE 的右边窗口看到现在所有剩余的证明目标。例如，你现在可以看到以下证明目标：

```
  G : GroupOperator
  GP : GroupProperties G
  ============================
  forall x : carrier_set, x + - x = 0
```

这里横线上方的是目前可以使用的前提，横线下方的是目前要证明的结论。接下去的每一行都是一条证明指令（tactic），每条证明指令可以将一个证明目标规约为 0 个，1 个或者更多的证明目标。

```
  intros x.
```

下面的 rewrite 指令可以把待证明结论中的项替换为与一个等价的项。在下面的第一条 rewrite 指令表示将`left_unit`性质的第一个参数代入为`x + (- x)`后进行替换。因此，待证明结论就变为了`0 + (x + (- x)) = 0`。

```
  rewrite <- (left_unit (x + (- x))).
```

下面这条指令中，通过 rewrite 指令后的箭头表明了替换的方向为使用`left_inv`等式的左侧替换该等式的右侧。而指令最后的 at 1 则表示只对第一个可以替换处进行替换。

```
  rewrite <- (left_inv (- x)) at 1.
```

在没有歧义的情况下，并不一定需要填写参数。

```
  rewrite -> assoc.
  rewrite <- (assoc (- x)).
```

如果不使用箭头，则默认为使用向右的箭头。

```
  rewrite left_inv.
  rewrite left_unit.
  rewrite left_inv.
```

最后，reflexivity 指令说的是：等式具有自反性，现在等式两边完全相同，所以已经证明完了。

```
  reflexivity.
Qed.
```

下面可以进一步证明右单位元性质。

```
Theorem right_unit {G: GroupOperator} {GP: GroupProperties G}:
  forall (x: carrier_set), x + 0 = x.
Proof.
  intros.
  rewrite <- (left_inv x).
  rewrite <- assoc.
```

证明中可以使用先前证明的结论。

```
  rewrite right_inv.
  rewrite left_unit.
  reflexivity.
Qed.
```

我们还可以证明双重取逆符号的消去。

```
Theorem inv_involutive {G: GroupOperator} {GP: GroupProperties G}:
  forall (x: carrier_set), - - x = x.
Proof.
  intros.
  rewrite <- (left_unit (- - x)).
  rewrite <- (right_inv x).
  rewrite assoc.
```

在 Coq 中，也可以将多条 rewrite 指令，写到一起。

```
  rewrite right_inv, right_unit.
  reflexivity.
Qed.
```

```
End Group.
```

2 归纳类型，递归定义与归纳证明

2.1 二叉查找树

```
Module BinarySearchTree0.
```

```
Inductive tree: Type :=
| Leaf: tree
| Node (l: tree) (v: Z) (r: tree): tree.
```

这个定义说的是，一棵二叉树要么是一棵空树`Leaf`，要么有一棵左子树、有一棵右子树外加有一个根节点整数标号。Coq 中，我们往往可以使用递归函数定义归纳类型元素的性质。Coq 中定义递归函数时使用的关键字是`Fixpoint`。下面的两个定义通过递归定义了二叉树的高度和节点个数。

```
Fixpoint tree_height (t: tree): Z :=
  match t with
  | Leaf => 0
  | Node l v r => Z.max (tree_height l) (tree_height r) + 1
  end.
```

```
Fixpoint tree_size (t: tree): Z :=
  match t with
  | Leaf => 0
  | Node l v r => tree_size l + tree_size r + 1
  end.
```

Coq 中也可以定义树到树的函数。下面的`tree_reverse`函数把二叉树进行了左右翻转。

```
Fixpoint tree_reverse (t: tree): tree :=
  match t with
  | Leaf => Leaf
  | Node l v r => Node (tree_reverse r) v (tree_reverse l)
  end.
```

下面画出的是一个二叉树左右翻转的例子。如果`t`是左边的树，那么`tree_reverse t`的计算结果就是右边的树。

```
      5                5
     / \              / \
    3   9            9   3
       / \          / \
      8  100      100  8
```

这个例子中的树以及左右翻转的计算结果都可以在 Coq 中表示出来：

Coq 表示

```
Example tree_reverse_example:
  tree_reverse
    (Node
       (Node Leaf 3 Leaf)
       5
       (Node (Node Leaf 8 Leaf) 9 (Node Leaf 100 Leaf)))
  =
  Node
    (Node (Node Leaf 100 Leaf) 9 (Node Leaf 8 Leaf))
    5
    (Node Leaf 3 Leaf).
Proof. reflexivity. Qed.
```

我们接下去将证明一些关于`tree_height`，`tree_size`与`tree_reverse`的基本性质。我们在证明中将会使用的主要方法是归纳法。相信大家都很熟悉自然数集上的数学归纳法。数学归纳法说的是：如果我们要证明某性质`P`对于任意自然数`n`都成立，那么我可以将证明分为如下两步：

- 奠基步骤：证明`P 0`成立；
- 归纳步骤：证明对于任意自然数`n`，如果`P n`成立，那么`P (n + 1)`也成立。

对二叉树的归纳证明与上面的数学归纳法稍有不同。具体而言，如果我们要证明某性质`P`对于一切二叉树`t`都成立，那么我们只需要证明以下两个结论：

- 奠基步骤：证明`P Leaf`成立；
- 归纳步骤：证明对于任意二叉树`l` `r`以及任意整数标签`n`，如果`P l`与`P r`都成立，那么`P (Node l n r)`也成立。

这样的证明方法就成为结构归纳法。在 Coq 中，`induction`指令表示：使用结构归纳法。下面是几个证明的例子。

第一个例子是证明`tree_size`与`tree_reverse`之间的关系。

```
Lemma reverse_size: forall t,
  tree_size (tree_reverse t) = tree_size t.
Proof.
  intros.
  induction t.
```

上面这个指令说的是：对`t`结构归纳。Coq 会自动将原命题规约为两个证明目标，即奠基步骤和归纳步骤。为了增加 Coq 证明的可读性，我们推荐大家使用 bullet 记号把各个子证明过程分割开来，就像一个一个抽屉或者一个一个文件夹一样。Coq 中可以使用的 bullet 标记有：`+ - * ++ -- **` ...

```
  + simpl.
```

第一个分支是奠基步骤。这个`simpl`指令表示将结论中用到的递归函数根据定义化简。

```
    reflexivity.
  + simpl.
```

第二个分支是归纳步骤。我们看到证明目标中有两个前提`IHt1`以及`IHt2`。在英文中`IH`表示 induction hypothesis 的缩写，也就是归纳假设。在这个证明中`IHt1`与`IHt2`分别是左子树`t1`与右子树`t2`的归纳假设。

```
    rewrite IHt1.
    rewrite IHt2.
    lia.
```

这个`lia`指令的全称是 linear integer arithmatic，可以用来自动证明关于整数的线性不等式。

```
Qed.
```

第二个例子很类似，是证明`tree_height`与`tree_reverse`之间的关系。

```
Lemma reverse_height: forall t,
  tree_height (tree_reverse t) = tree_height t.
Proof.
  intros.
  induction t.
  + simpl.
    reflexivity.
  + simpl.
    rewrite IHt1.
    rewrite IHt2.
    lia.
```

注意：这个`lia`指令也是能够处理`Z.max`与`Z.min`的。

```
Qed.
```

下面我们将通过重写上面这一段证明，介绍 Coq 证明语言的一些其他功能。

```
Lemma reverse_height_attempt2: forall t,
  tree_height (tree_reverse t) = tree_height t.
Proof.
  intros.
  induction t; simpl.
```

在 Coq 证明语言中可以用分号将小的证明指令连接起来形成大的证明指令，其中`tac1 ; tac2`这个证明指令表示先执行指令`tac1`，再对于`tac1`生成的每一个证明目标执行`tac2`。分号是右结合的。

```
  + reflexivity.
  + simpl.
    lia.
```

此处的`lia`指令不仅可以处理结论中的整数线性运算，其自动证明过程中也会使用前提中关于整数线性运算的假设。

```
Qed.
```

请各位同学证明下面结论。

```
Lemma reverse_involutive: forall t,
  tree_reverse (tree_reverse t) = t.
Proof.
Admitted. (* WORK IN CLASS *)
```

提示：下面的结论可以不用归纳法证明。

```
Lemma reverse_inv: forall t1 t2,
  tree_reverse t1 = t2 ->
  t1 = tree_reverse t2.
Proof.
Admitted. (* WORK IN CLASS *)
```

下面两个小题留作习题。

```
Lemma size_nonneg: forall t,
  0 <= tree_size t.
Proof.
Admitted. (* 留作习题 *)
```

```
Fixpoint left_most (t: tree) (default: Z): Z :=
  match t with
  | Leaf => default
  | Node l n r => left_most l n
  end.

Fixpoint right_most (t: tree) (default: Z): Z :=
  match t with
  | Leaf => default
  | Node l n r => right_most r n
  end.

Lemma left_most_reverse: forall t default,
  left_most (tree_reverse t) default = right_most t default.
Proof.
  intros t.
Admitted. (* 留作习题 *)
```

```
End BinarySearchTree0.
```

2.2 自然数

在 Coq 中，许多数学上的集合可以用归纳类型定义。例如，Coq 中自然数的定义就是最简单的归纳类型之一。

- 下面 Coq 代码可以用于查看`nat`在 Coq 中的定义。

  ```
  Print nat.
  ```

- 查询结果如下。

```
Inductive nat := O : nat | S: nat -> nat.
```

下面我们在 Coq 中去定义自然数的加法，并且也试着证明一条基本性质：加法交换律。

由于 Coq 的标准库中已经定义了自然数以及自然数的加法。我们开辟一个`NatDemo`来开发我们自己的定义与证明。以免与 Coq 标准库的定义相混淆。

```
Module NatDemo.
```

先定义自然数`nat`。

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

再定义自然数加法。

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

下面证明加法交换律。

```
Theorem add_comm: forall n m,
  add n m = add m n.
Proof.
  intros.
  induction n.
```

证明到此处，我们发现我们需要首先证明`n + 0 = n`这条性质，我们先终止交换律的证明，而先证明这条引理。

```
Abort.
```

```
Lemma add_O_r: forall n, add n O = n.
Proof.
  intros.
  induction n; simpl.
  + reflexivity.
  + rewrite IHn.
    reflexivity.
Qed.
```

```
Theorem add_comm: forall n m,
  add n m = add m n.
Proof.
  intros.
  induction n; simpl.
  + rewrite add_O_r.
    reflexivity.
  +
```

证明到此处，我们发现我们需要还需要证明关于`m + (S n)`相关的性质。

```
Abort.
```

```
Lemma add_S_r: forall n m,
  add n (S m) = S (add n m).
Proof.
  intros.
  induction n; simpl.
  + reflexivity.
  + rewrite IHn.
    reflexivity.
Qed.
```

现在已经可以在 Coq 中完成加法交换律的证明了。

```
Theorem add_comm: forall n m,
  add n m = add m n.
Proof.
  intros.
  induction n; simpl.
  + rewrite add_0_r.
    reflexivity.
  + rewrite add_S_r.
    rewrite IHn.
    reflexivity.
Qed.
```

```
End NatDemo.
```

上面证明的加法交换律在 Coq 标准库中已有证明，其定理名称是`Nat.add_comm`。

```
Check Nat.add_comm.
```

2.3 例子：结合律、单位元与幂运算

```
Module AssocUnit.
```

下面我们结合先前所学，来证明一些关于结合律、单位元和幂运算的性质。在数学上，如果一个二元运算有结合律，那么在书写时就可以省略括号，忽略结合顺序。更进一步，在结合律的基础上可以引入幂运算。例如，`a ○ (a ○ a) = (a ○ a) ○ a`，所以我们可以用`a`的立方来表示这个等式两边的值。下面我们在 Coq 中定义这些概念并且证明一些性质。

```
Class AssocUnitOperator: Type := {
  carrier_set: Type;
  unit: carrier_set;
  op: carrier_set -> carrier_set -> carrier_set
}.
```

我们同样引入一些 Notation 来表示此处我们讨论的单位元与二元运算。

```
Notation "1" := (unit).
Notation "x ○ y" := (op x y) (left associativity, at level 61).
```

我们假设该二元运算符合结合律、左单位元性质与右单位元性质。

```
Class AssocUnitProperties (AU: AssocUnitOperator): Type := {
  assoc: forall (x y z: carrier_set), (x ○ y) ○ z = x ○ (y ○ z);
  left_unit: forall (x: carrier_set), 1 ○ x = x;
  right_unit: forall (x: carrier_set), x ○ 1 = x
}.
```

在 Coq 中，可以用 Section 与 Context 关键字统一引入一下假设与前提，这些假设与前提会在 End 处失效。

```
Section AssocUnit.

Context {AU: AssocUnitOperator}.
Context {AUP: AssocUnitProperties AU}.
```

幂运算可以在 Coq 中对自然数归纳定义：

```
Fixpoint power (x: carrier_set) (n: nat): carrier_set :=
  match n with
  | O => unit
  | S n' => power x n' ○ x
  end.
```

根据上面定义，`x` 就是 `x` 本身的一次幂。

```
Lemma power_1: forall x, x = power x 1.
Proof.
  intros.
  simpl.
  rewrite left_unit.
  reflexivity.
Qed.
```

`x ○ x` 就是 `x` 本身的二次幂。

```
Lemma power_2: forall x, x ○ x = power x 2.
Proof.
  intros.
  simpl.
  rewrite left_unit.
  reflexivity.
Qed.
```

下面证明幂运算的性质：

```
Lemma power_mul_power:
  forall x n m, power x n ○ power x m = power x (n + m).
Proof.
  intros.
  revert n; induction m; intros; simpl.
  + rewrite right_unit.
    rewrite Nat.add_0_r.
    reflexivity.
  + rewrite Nat.add_succ_r.
    simpl.
    rewrite <- assoc.
    rewrite IHm.
    reflexivity.
Qed.
```

下面这条性质刻画了幂运算的幂运算的性质。请同学们试着完成证明，提示：证明中可能会用到 `Nat.add_comm`。

```
Lemma power_power:
  forall x n m, power (power x m) n = power x (n * m).
Proof.
  intros.
Admitted. (* WORK IN CLASS *)
```

```
End AssocUnit.
End AssocUnit.
```

3 命题与集合

在 Coq 中还可以用与、或、非、如果-那么、存在以及任意来描述复杂的命题，并且证明相关性质。

3.1 逻辑命题『真』的证明

我们不需要任何前提就可以推出`True`。在 Coq 标准库中，`I`是`True`的一个证明，我们可以用`exact I`来证明`True`。

```
Example proving_True_1: 1 < 2 -> True.
Proof.
  intros.
  exact I.
Qed.

Example proving_True_2: 1 > 2 -> True.
Proof.
  intros.
  exact I.
Qed.
```

3.2 关于『并且』的证明

要证明『某命题并且某命题』成立，可以在 Coq 中使用`split`证明指令进行证明。该指令会将当前的证明目标拆成两个子目标。

```
Lemma True2: True /\ True.
Proof.
  split.
  + exact I.
  + exact I.
Qed.
```

下面证明一个关于`/\`的一般性结论：

```
Lemma and_intro : forall A B : Prop, A -> B -> A /\ B.
Proof.
  intros A B HA HB.
  split.
```

下面的`apply`指令表示在证明中使用一条前提，或者使用一条已经经过证明的定理或引理。

```
  + apply HA.
  + apply HB.
Qed.

Example and_exercise :
  forall n m : Z, n + 2*m = 10 -> 2*n + m = 5 -> n = 0 /\ True.
Proof.
Admitted. (* WORK IN CLASS *)
```

如果当前一条前提假设具有『某命题并且某命题』的形式，我们可以在 Coq 中使用 `destruct` 指令将其拆分成为两个前提。

```
Lemma proj1 : forall P Q : Prop,
  P /\ Q -> P.
Proof.
  intros.
  destruct H as [HP HQ].
  apply HP.
Qed.
```

`destruct` 指令也可以不指名拆分后的前提的名字，Coq 会自动命名。

```
Lemma proj2 : forall P Q : Prop,
  P /\ Q -> Q.
Proof.
  intros.
  destruct H.
  apply H0.
Qed.
```

当前提与结论中，都有 `/\` 的时候，我们就既需要使用 `split` 指令，又需要使用 `destruct` 指令。

```
Theorem and_commut : forall P Q : Prop,
  P /\ Q -> Q /\ P.
Proof.
  intros.
  destruct H as [HP HQ].
  split.
  - apply HQ.
  - apply HP.
Qed.

Theorem and_assoc : forall P Q R : Prop,
  P /\ (Q /\ R) -> (P /\ Q) /\ R.
Proof.
Admitted. (* WORK IN CLASS *)
```

3.3 关于『或』的证明

『或』是另一个重要的逻辑连接词。如果『或』出现在前提中，我们可以用 Coq 中的 `destruct` 指令进行分类讨论。

```
Lemma or_example :
  forall n m : Z, n = 0 \/ m = 0 -> n * m = 0.
Proof.
  intros.
  destruct H as [H | H].
  + rewrite H.
    lia.
  + rewrite H.
    lia.
Qed.
```

在上面的例子中，我们对于形如`A \/ B`的前提进行分类讨论。要证明`A \/ B`能推出原结论，就需要证明`A`与`B`中的任意一个都可以推出原结论。下面是一个一般性的结论.

```
Lemma or_example2 :
  forall P Q R: Prop, (P -> R) -> (Q -> R) -> (P \/ Q -> R).
Proof.
  intros.
  destruct H1 as [HP | HQ].
  + apply H.
```

注意，`apply`指令不一定要前提与结论完全吻合才能使用。此处，只要`H`中推导的结果与待证明的结论一致，就可以使用`apply H`。

```
    apply HP.
  + apply H0 in HQ.
```

`apply`指令还可以在前提中做推导，不过这时需要使用`apply ... in`这一语法。

```
    apply HQ.
Qed.
```

相反的，如果要证明一条形如`A \/ B`的结论整理，我们就只需要证明`A`与`B`两者之一成立就可以了。在 Coq 中的指令是：`left`与`right`。例如，下面是选择左侧命题的例子。

```
Lemma or_introl : forall A B : Prop, A -> A \/ B.
Proof.
  intros.
  left.
  apply H.
Qed.
```

下面是选择右侧命题的例子。

```
Lemma or_intror : forall A B : Prop, B -> A \/ B.
Proof.
  intros.
  right.
  apply H.
Qed.
```

下面性质请各位自行证明。

```
Theorem or_commut : forall P Q : Prop,
  P \/ Q  -> Q \/ P.
Proof.
Admitted. (* WORK IN CLASS *)
```

3.4 关于『如果... 那么...』的证明

事实上，在之前的例子中，我们已经多次证明有关`->`的结论了。下面我们在看几个例子，并额外介绍几条 Coq 证明指令。

下面的证明中，`pose proof`表示推导出一个新的结论，并将其用作之后证明中的前提。

```
Theorem modus_ponens: forall P Q: Prop,
  P /\ (P -> Q) -> Q.
Proof.
  intros.
  destruct H.
```

将`H0: P -> Q`作用在`H: P`上，我们就可以得出一个新结论：`Q`。

```
  pose proof H0 H.
  apply H1.
Qed.
```

下面我们换一种方法证明。`revert`证明指令可以看做`intros`的反操作。

```
Theorem modus_ponens_alter1: forall P Q: Prop,
  P /\ (P -> Q) -> Q.
Proof.
  intros.
  destruct H.
```

下面`revert`指令将前提中的`P`又放回了『结论中的前提』中去。

```
  revert H.
  apply H0.
Qed.
```

下面我们再换一种方式证明，`specialize`指令与`apply ... in`指令的效果稍有不同。

```
Theorem modus_ponens_alter2: forall P Q: Prop,
  P /\ (P -> Q) -> Q.
Proof.
  intros.
  destruct H.
  specialize (H0 H).
  apply H0.
Qed.
```

另外，我们可以直接使用`exact`指令，这个指令的效果像是`pose proof`或者`specialize`与`apply`的组合。

```
Theorem modus_ponens_alter3: forall P Q: Prop,
  P /\ (P -> Q) -> Q.
Proof.
  intros.
  destruct H.
  exact (H0 H).
Qed.
```

3.5 关于『否定』与『假』的证明

在 Coq 中 [ ] 表示否定，`False`表示假。如果前提为假，那么，矛盾推出一切。在 Coq 中，这可以用`contradiction`指令或`destruct`指令完成证明。

```
Theorem ex_falso_quodlibet : forall (P: Prop),
  False -> P.
Proof.
  intros.
  contradiction.
Qed.
```

```
Theorem ex_falso_quodlibet_alter : forall (P: Prop),
  False -> P.
Proof.
  intros.
  destruct H.
Qed.
```

`contradiction`也可以用于`P`与`~ P`同时出现在前提中的情况：

```
Theorem contradiction_implies_anything : forall P Q : Prop,
  (P /\ ~ P) -> Q.
Proof.
  intros.
  destruct H.
  contradiction.
Qed.
```

除了`P`与`~ P`不能同时为真之外，他们也不能同时为假，或者说，他们中至少有一个要为真。这是 Coq 标准库中的`classic`。

```
Check classic.
```

它说的是：`forall P : Prop, P \/ ~ P`。下面我们利用它做一些证明。

```
Theorem double_neg_elim : forall P : Prop,
  ~ ~ P -> P.
Proof.
  intros.
  pose proof classic P.
  destruct H0.
  + apply H0.
  + contradiction.
Qed.
```

```
Theorem not_False :
  ~ False.
Proof.
Admitted. (* WORK IN CLASS *)
```

```
Theorem double_neg_intro : forall P : Prop,
  P -> ~ ~ P.
Proof.
Admitted. (* WORK IN CLASS *)
```

3.6 关于『当且仅当』的证明

在 Coq 中，`<->`符号对应的定义是`iff`，其将`P <-> Q`定义为`(P -> Q) /\ (Q -> P)`因此，要证明关于『当且仅当』的性质，首先可以使用其定义进行证明。

```
Theorem iff_refl: forall P: Prop, P <-> P.
Proof.
  intros.
  unfold iff.
  split.
  + intros.
    apply H.
  + intros.
    apply H.
Qed.
```

```
Theorem iff_imply: forall P Q: Prop, (P <-> Q) -> (P -> Q).
Proof.
  intros P Q H.
  unfold iff in H.
  destruct H.
  exact H.
Qed.
```

当某前提假设具有形式`P <-> Q`，那我们也可以使用`apply`指令进行证明。

```
Theorem iff_imply_alter: forall P Q: Prop, (P <-> Q) -> (P -> Q).
Proof.
  intros.
  apply H.
  apply H0.
Qed.
```

另外，`rewrite`指令也可以使用形如`P <-> Q`的等价性前提。

```
Theorem iff_imply_alter2: forall P Q: Prop, (P <-> Q) -> (P -> Q).
Proof.
  intros.
  rewrite <- H.
  apply H0.
Qed.
```

3.7 关于『存在』的证明

当待证明结论形为：存在一个`x`使得...，那么可以用`exists`指明究竟哪个`x`使得该性质成立。

```
Lemma four_is_even : exists n, 4 = n + n.
Proof.
  exists 2.
  lia.
Qed.
```

```
Lemma six_is_not_prime: exists n, 2 <= n < 6 /\ exists q, n * q = 6.
Proof.
  exists 2.
  split.
  + lia.
  + exists 3.
    lia.
Qed.
```

当某前提形为：存在一个`x`使得...，那么可以使用 Coq 中的`destruct`指令进行证明。这一证明指令相当于数学证明中的：任意给定一个这样的`x`。

```
Theorem exists_example : forall n,
  (exists m, n = 4 + m) ->
  (exists o, n = 2 + o).
Proof.
  intros.
  destruct H as [m H].
  exists (2 + m).
  lia.
Qed.
```

```
Theorem dist_exists_and : forall (X: Type) (P Q: X -> Prop),
  (exists x, P x /\ Q x) -> (exists x, P x) /\ (exists x, Q x).
Proof.
Admitted. (* WORK IN CLASS *)
```

```
Theorem exists_exists : forall (X Y: Type) (P: X -> Y -> Prop),
  (exists x y, P x y) <-> (exists y x, P x y).
Proof.
Admitted. (* WORK IN CLASS *)
```

3.8 关于『任意』的证明

关于『任意』的证明与关于『如果... 那么...』的证明是类似的，我们可以灵活使用`pose proof`，`specialize`，`apply`，`revert`等指令进行证明。下面是一个简单的例子。

```
Theorem forall_forall : forall (X Y: Type) (P: X -> Y -> Prop),
  (forall x y, P x y) -> (forall y x, P x y).
Proof.
  intros X Y P H.
  intros y x.
  specialize (H x y).
  apply H.
Qed.
```

3.9 关于命题逻辑的自动证明

除了上述证明指令之外，Coq 还提供了`tauto`这一自动证明指令。如果当且结论可以完全通过命题逻辑完成证明，那么`tauto`就可以自动构造这样的证明。例如：

```
Lemma or_reduce: forall (P Q R: Prop),
  (P /\ R) \/ (Q /\ ~ R) -> P \/ Q.
Proof.
  intros.
  tauto.
Qed.
```

这里，所谓完全通过命题逻辑完成证明，指的是通过对于『并且』、『或』、『非』、『如果... 那么...』、『当且仅当』、『真』与『假』的推理完成证明（注：除此之外，`tauto`也会将形如`a = a`的命题看做`True`）。下面例子所描述的命题很符合直觉，但是它就不能仅仅使用命题逻辑完成推理，因为他的推理过程中用到了`forall`的性质。

```
Lemma forall_and: forall (A: Type) (P Q: A -> Prop),
  (forall a: A, P a /\ Q a) <-> (forall a: A, P a) /\ (forall a: A, Q a).
Proof.
  intros.
```

注意，此处不能使用`tauto`直接完成证明。

```
  split.
  + intros.
    split.
    - intros a.
      specialize (H a).
```

此时可以用`tauto`完成剩余证明了，另外两个分支也是类似。

```
      tauto.
    - intros a.
      specialize (H a).
      tauto.
  + intros.
    destruct H.
    specialize (H a).
    specialize (H0 a).
    tauto.
Qed.
```

3.10 集合性质的证明

在 Coq 中往往使用`X: A -> Prop`来表示某类型`A`中元素构成的集合`X`。字面上看，这里的`A -> Prop`表示`X`是一个从`A`中元素到命题的映射，这也相当于说`X`是一个关于`A`中元素性质。对于每个`A`中元素`a`而言，`a`符合该性质`X`等价于`a`对应的命题`X a`为真，又等价于`a`是集合`X`的元素。

```
Module SetsProperties.
Local Open Scope sets.
```

下面这个命题描述了整数集合之间交集的交换律。

```
Lemma Sets_intersect_comm: forall (X Y: Z -> Prop),
  X ∩ Y == Y ∩ X.
Proof.
  intros.
```

证明时，`sets_unfold`指令可以将关于集合的性质转化为关于逻辑的性质。

```
  sets_unfold.
```

命题转化为，对于任意一个整数`a`，`X a`为真且`Y a`为真当且仅当`Y a`为真且`X a`为真。

```
  intros.
  tauto.
Qed.

Lemma Sets_union_comm: forall (X Y: Z -> Prop),
  X ∪ Y == Y ∪ X.
Proof.
Admitted. (* WORK IN CLASS *)

Lemma Sets_union_assoc: forall (X Y Z: Z -> Prop),
  (X ∪ Y) ∪ Z == X ∪ (Y ∪ Z).
Proof.
Admitted. (* WORK IN CLASS *)
```

```
End SetsProperties.
```

SetsDomain 库中提供了一系列有关集合运算的性质的证明。未来大家在证明中既可以使用`sets_unfold`将关于集合运算的命题转化为关于逻辑的命题，也可以直接使用下面这些性质完成证明。

SetsDomain 中提供的引理（部分）

```
  Sets_equiv_Sets_included:
    forall x y, x == y <-> x ⊆ y /\ y ⊆ x;
  Sets_empty_included:
    forall x, ∅ ⊆ x;
  Sets_included_full:
    forall x, x ⊆ Sets.full;
  Sets_intersect_included1:
    forall x y, x ∩ y ⊆ x;
  Sets_intersect_included2:
    forall x y, x ∩ y ⊆ y;
  Sets_included_intersect:
    forall x y z, x ⊆ y -> x ⊆ z -> x ⊆ y ∩ z;
  Sets_included_union1:
    forall x y, x ⊆ x ∪ y;
  Sets_included_union2:
    forall x y, y ⊆ x ∪ y;
  Sets_union_included_strong2:
    forall x y z u, x ∩ u ⊆ z -> y ∩ u ⊆ z -> (x ∪ y) ∩ u ⊆ z;
  Sets_included_omega_union:
    forall xs n, xs n ⊆ ⋃ xs;
  Sets_omega_union_included:
    forall xs y, (forall n, xs n ⊆ y) -> ⋃ xs ⊆ y;
  Sets_omega_intersect_included:
    forall xs n, ⋂ xs ⊆ xs n;
  Sets_included_omega_intersect:
    forall xs y, (forall n : nat, y ⊆ xs n) -> y ⊆ ⋂ xs;
```

3.11 例子：二叉查找树表达的集合

在程序开发的过程中，二叉查找树一般用于表示一个集合或者表示一个键（Key）到值（Value）的映射，并且实现高效的插入、删除与查找。前面我们定义的二叉查找树就可以描述一个值的集合。

```
Module BinarySearchTree1.
Import BinarySearchTree0.
Local Open Scope sets.
```

下面定义中，`denote t`就定义了二叉查找树`t`所描述的值的集合。

```
Fixpoint denote (t: tree): Z -> Prop :=
  match t with
  | Leaf => [ ]
  | Node l a r => denote l ∪ [ a ] ∪ denote r
  end.
```

基于此，我们容易证明，二叉查找树的左旋与右旋变化不改变二叉查找树所描述的值的集合。

```
Lemma rotate_sound: forall t1 x t2 y t3,
  denote (Node t1 x (Node t2 y t3)) ==
  denote (Node (Node t1 x t2) y t3).
Proof.
  intros.
  simpl.
  sets_unfold.
  intros.
  tauto.
Qed.
```

```
End BinarySearchTree1.
```

4 二元关系、算法描述与算法验证

```
Local Open Scope sets.
```

4.1 二元关系

在数学上，二元关系是一种集合，在 Coq 中，我们用`A -> B -> Prop`表示`A`与`B`之间的二元关系。下面定义一种特殊的二元关系『相等关系』与一种二元关系之间的运算『连接』运算。之后，我们就可以用`BinRel.id`与`BinRel.concat`来指代它们。

```
Module BinRel.

Definition id {A: Type}: A -> A -> Prop := fun a b => a = b.

Definition concat
             {A B C: Type}
             (r1: A -> B -> Prop)
             (r2: B -> C -> Prop): A -> C -> Prop :=
  fun a c => exists b, r1 a b /\ r2 b c.

End BinRel.
```

在此基础上，就可以定义二元关系的自反传递闭包。下面定义的`nsteps x n a b`表示从状态`a`出发可以沿着二元关系`x`走`n`步到达`b`。

```
Fixpoint nsteps
           {A: Type}
           (x: A -> A -> Prop) (n: nat): A -> A -> Prop :=
  match n with
  | O => BinRel.id
  | S n' => BinRel.concat (nsteps x n') x
  end.
```

下面定义的`clos_refl_trans x a b`表示从状态`a`出发可以沿着二元关系`x`走有穷多步到达`b`。

```
Definition clos_refl_trans
           {A: Type}
           (x: A -> A -> Prop): A -> A -> Prop :=
  ⋃ (nsteps x).
```

```
Local Close Scope sets.
```

下面我们将使用二元关系来描述算法，并证明算法的正确性。一般而言，可以用一个二元关系描述算法步骤，用它的自反传递闭包来描述整个算法。

4.2 例子：快速幂

```
Module FastPower.
Import AssocUnit.
Local Open Scope nat.
Local Open Scope sets.
```

```
Section FastPower.
```

快速幂算法的伪代码非常简单：

```
result = unit;
while (n > 0) {
  if (n % 2 == 1) {
    result = result * base;
  }
  n = n / 2;
  base = base * base;
}
```

它适用于一切具有结合律与单位元的二元运算。在刻画该算法并证明其正确性时，我们可以引用先前定义的相关假设。

```
Context {AU: AssocUnitOperator}.
Context {AUP: AssocUnitProperties AU}.
```

下面我们使用 Coq 中的 Record 来定义这一算法的程序状态。一个程序状态`s`包含 3 部分：指数`s.(exponent)`（亦可在 Coq 中写作`exponent s`，表示上面伪代码中变量`n`的值）；底数`s.(base)`（亦可在 Coq 中写作`base s`，表示上面伪代码中变量`base`的值）；`s.(result)`（亦可在 Coq 中写作`result s`，表示上面伪代码中变量`result`的值）。

```
Record state: Type := {
  exponent: nat;
  base: carrier_set;
  result: carrier_set
}.
```

执行一次循环体可能发生下面两种程序行为之一。`step_odd`表述了伪代码中变量`n`为奇数的情况，`step_even`表述了伪代码中`n`为偶数的情况。

```
Record step_odd (s1 s2: state): Prop := {
  step_odd_exp: s1.(exponent) = s2.(exponent) * 2 + 1;
  step_odd_base: s2.(base) = s1.(base) ∘ s1.(base);
  step_odd_result: s2.(result) = s1.(result) ∘ s1.(base)
}.

Record step_even (s1 s2: state): Prop := {
  step_even_exp: s1.(exponent) = s2.(exponent) * 2;
  step_even_base: s2.(base) = s1.(base) ∘ s1.(base);
  step_even_result: s2.(result) = s1.(result)
}.
```

上面定义说的是：`step_odd s1 s2` 成立当且仅当其定义中的三条性质都成立。在证明中，假如 `H: step_odd s1 s2`，那么可以使用 `step_odd_exp _ _ H` 这样的方式得到其定义中的第一条性质，其余情况依次类推。下面我们将分 3 步完成证明。第一步，单个循环体能保持不变式成立。

```
Lemma step_invariant: forall s1 s2,
  (step_odd ∪ step_even) s1 s2 ->
  s2.(result) ∘ power s2.(base) s2.(exponent) =
  s1.(result) ∘ power s1.(base) s1.(exponent).
Proof.
  intros.
  destruct H.
  + rewrite (step_odd_exp _ _ H).
    rewrite (step_odd_base _ _ H).
    rewrite (step_odd_result _ _ H).
    rewrite (power_1 s1.(base)) at 1.
    rewrite (power_2 s1.(base)).
    rewrite power_power.
    rewrite assoc.
    rewrite power_mul_power.
    rewrite Nat.add_comm.
    reflexivity.
  + rewrite (step_even_exp _ _ H).
    rewrite (step_even_base _ _ H).
    rewrite (step_even_result _ _ H).
    rewrite (power_2 s1.(base)).
    rewrite power_power.
    reflexivity.
Qed.
```

第二步：整个循环保持不变式成立。

```
Lemma multi_step_invariant: forall s1 s2,
  clos_refl_trans (step_odd ∪ step_even) s1 s2 ->
  s2.(result) ∘ power s2.(base) s2.(exponent) =
  s1.(result) ∘ power s1.(base) s1.(exponent).
Proof.
  intros.
  unfold clos_refl_trans in H.
  destruct H as [n ?].
  revert s2 H.
  induction n; intros.
  + simpl in H.
    rewrite <- H.
    reflexivity.
  + destruct H as [s2' [? ?]].
    apply IHn in H; clear IHn.
    apply step_invariant in H0.
    rewrite <- H, H0.
    reflexivity.
Qed.
```

第三步：基于不变式证明算法正确性。

```
Theorem fast_power_sound: forall s1 s2,
  clos_refl_trans (step_odd ∪ step_even) s1 s2 ->
  s2.(exponent) = 0 ->
  s1.(result) = unit ->
  s2.(result) = power s1.(base) s1.(exponent).
Proof.
  intros.
  apply multi_step_invariant in H.
  rewrite H0, H1 in H; clear H0 H1.
  simpl in H.
  rewrite left_unit, right_unit in H.
  apply H.
Qed.
```

```
End FastPower.
```