简单 Coq 证明与定义

1 整数算数运算与大小比较

算术与数量之间的大小关系是日常生活中最常见的数学概念。我们可以用数描述物体的数量、几何图形的长度与面积、人的年龄等等。作为 Coq 验证工具的入门篇，这一章将介绍如何使用 Coq 表达一些简单的数量关系，并证明一些简单的性质。

大约在 1500 年前，《孙子算经》一书中记载了这样一个有趣的问题：今有雉兔同笼，上有三十五头，下有九十四足，问雉兔各几何？这就是著名的鸡兔同笼问题。我们都知道，如果用 C 表示鸡（Chicken）的数量，用 R 表示兔（Rabbit）的数量，那么这一问题中的数量关系就可以表示为：

```
C + R = 35
2 * C + 4 * R = 94
```

进而可以求解得知 `C = 23` （也可以求解得到 `R = 12` ）。这一推理过程可以在 Coq 中表示成为下面命题。

```
Fact chickens_and_rabbits: forall C R: Z,
  C + R = 35 ->
  2 * C + 4 * R = 94 ->
  C = 23.
```

字面意思上，这个命题说的是：对于任意整数 `C` 与 `R` ，如果 `C + R = 35` 并且 `2 * C + 4 * R = 94` ，那么 `C = 23` 。其中， `forall` 与 `->` 是 Coq 中描述数学命题的常用符号。

Coq 表达式 1. 全称量词 forall . 在 Coq 中， `forall` 表示“任意”的意思，例如：

```
forall x: Z, x = x
```

就是一个语法上合法的 Coq 命题，在这个例子中 `Z` 表示整数集合， `forall x: Z, ...` 说的就是“对于任意整数 `x` ，某性质成立”。在 `forall` 之后，可以跟一个变量也可以跟多个变量，例如：

```
forall x y: Z, x + y = y + x
```

也是合法的 Coq 命题。另外， `forall` 后的类型标注不是必须的，如果 Coq 系统能够推导出这个类型，那么就可以省略它。Coq 还允许一些 `forall` 之后的变量有类型标注，而另一些没有，例如：

```
forall (x: Z) y, x + y = y + x
```

。

Coq 表达式 2. 表示命题推导的箭头符号 `->` . 在 Coq 中，箭头符号 `->` 表示“如果... 那么...”，例如：

```
x >= 0 -> x + 1 > 0
```

就表示如果 `x` 大于等于 0，那么 `x+1` 大于 0。Coq 规定，这个箭头符号是右结合的，换言之， `P1 -> P2 -> P3` 实际是 `P1 -> (P2 -> P3)` 的简写，其表达的意思是：如果 `P1` 成立，那么 `P2` 能推出 `P3` 。逻辑上，这等同于：如果 `P1` 并且 `P2` ，那么 `P3` 。因此，我们一般会将形如 `P1 -> P2 -> ... -> Pn -> Q` 的命题读作：如果 `P1` 、 `P2` 、...、 `Pn` 都成立，那么 `Q` 也成立。

上面的 Coq 代码中，除了逻辑符号 `forall` 、 `->` 与算数符号之外，还使用了保留字 `Fact` ，这是一种 Coq 指令。

Coq 指令 1. Fact、Proposition、Example、Lemma、Theorem 与 Corollary 指令. 在 Coq 中，Fact 指令可以用于陈述一个命题。例如，在

```
Fact chickens_and_rabbits: forall C R: Z,
  C + R = 35 ->
  2 * C + 4 * R = 94 ->
  C = 23.
```

上面这段代码中，`chickens_and_rabbits` 是命题的名字，之后从 `forall` 开头的逻辑算数表达式是这个命题的内容，命题的名字与命题的内容之间用冒号分隔。Coq 系统规定，只要这个命题语法上合法，那么整个 Fact 指令就是合法的。换言之，Coq 不会在执行 Fact 指令的时候检查其声明的命题是不是真命题。不过，执行 Fact 指令之后，用户需要进入 Coq 证明环境证明该结论。在 Coq 中，还有一些保留字与 Fact 功能相同，它们是：Proposition、Example、Lemma、Theorem 与 Corollary。

在 Fact 指令之后，我们可以在 Coq 中证明这个数学命题成立。如果要用中学数学知识完成这一证明，恐怕要使用加减消元法、代入消元法等代数技巧。Coq 并不需要我们掌握这些数学技巧，Coq 系统可以自动完成整数线性运算性质（linear integer arithmetic，简称 lia）的证明，`chickens_and_rabbits` 这一命题在 Coq 中的证明只需一行：

```
Proof. lia. Qed.
```

在这一行代码中，Proof 和 Qed 表示一段证明的开头与结尾，在它们之间的 `lia` 指令是证明脚本。

一般而言，编写 Coq 证明的过程是交互式的——“交互式”的意思是：在编写证明代码的同时，我们可以在 Coq 定理证明环境中运行证明脚本，获得反馈，让定理证明系统告诉我们“已经证明了哪些结论”、“还需要证明哪些结论”等等，并以此为依据编写后续的证明代码。安装 VSCoq 插件的 VSCode 编辑器、安装 proof-general 插件的 emacs 编辑器以及 CoqIDE 都是成熟易用的 Coq 定理证明环境。

以上面的证明为例，执行 `lia` 指令前，证明窗口显示了当前还需证明的性质（亦称为证明目标，proof goal）：

```
------------------------------------------
(1/1)
forall C R : Z,
C + R = 35 -> 2 * C + 4 * R = 94 -> C = 23
```

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，目前，前提集合为空。横线下方的 (1/1) 表示总共还有 1 个证明目标需要证明，当前显示的是其中的第一个证明目标。利用证明脚本证明的过程中，每一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。执行 `lia` 指令之后，证明窗口显示：Proof finished。表示证明已经完成。一般情况下，Coq 证明往往是不能只靠一条证明指令完成证明的。

Coq 指令 2. Proof 指令与 Qed 指令. Proof 与 Qed 是一段证明的首尾标识，在它们之间的 Coq 指令都是证明脚本。在 Coq 中，用户通过证明脚本完成证明。一般情况下，Coq 证明脚本都能保证其进行的逻辑变换与逻辑规约都是合法的，特殊情况下，Coq 定理证明系统还需要在 Qed 指令时进行额外检验。经过 Qed 检验后，一个数学命题的 Coq 证明才算完成。

Coq 证明脚本 1. lia 指令. 证明指令 `lia` 表示自动证明有关整数线性运算与大小关系的性质，lia 这三个字母是 linear integer arithmetic 的缩写。证明指令 `lia` 是完备的，换言之，所有正确的整数线性运算性质都能够通过这一指令设定的算法完成自动证明。当然，在实际使用中，可能由于待证明的命题规模太大（变量个数太多、约束条件中的表达式太长或约束条件数量太多），算法所需运行时间太长，Coq 系统将其提前终止，因而无法完成自动证明。

Coq 证明指令 `lia` 除了能够证明关于线性运算的等式，也可以证明关于线性运算的不等式。下面这个例子选自熊斌教授所著《奥数教程》的小学部分：幼儿园的小朋友去春游，有男孩、女孩、男老师与女老师共 16 人，已知小朋友比老师人数多，但是女老师比女孩人数多，女孩又比男孩人数多，男孩比男老师人数多，请证明幼儿园只有一位男老师。

```
Fact teachers_and_children: forall MT FT MC FC: Z,
  MT > 0 ->
  FT > 0 ->
  MC > 0 ->
  FC > 0 ->
  MT + FT + MC + FC = 16 ->
  MC + FC > MT + FT ->
  FT > FC ->
  FC > MC ->
  MC > MT ->
  MT = 1.
Proof. lia. Qed.
```

习题 1. 请在 Coq 中描述下面结论并证明：如果今年甲的年龄是乙 5 倍，并且 5 年后甲的年龄是乙的 3 倍，那么今年甲的年龄是 25 岁。

除了线性性质之外，Coq 中还可以证明的一些更复杂的性质。例如下面就可以证明，任意两个整数的平方和总是大于它们的乘积。证明中使用的指令 `nia` 表示的是非线性整数运算（nonlinear integer arithmetic）求解。

```
Fact sum_of_sqr1: forall x y: Z,
  x * x + y * y >= x * y.
Proof. nia. Qed.
```

不过，`nia` 与 `lia` 不同，后者能够保证关于线性运算的真命题总能被自动证明（规模过大的除外），但是有不少非线性的算数运算性质是 `nia` 无法自动求解的。例如，下面例子说明，一些很简单的命题 `nia` 都无法完成自动验证。

```
Fact sum_of_sqr2: forall x y: Z,
  x * x + y * y >= 2 * x * y.
Proof. Fail nia. Abort.
```

这时，我们就需要编写证明脚本，给出中间证明步骤。证明过程中，可以使用 Coq 标准库中的结论，也可以使用我们自己实现证明的结论。例如，Coq 标准库中，`sqr_pos` 定理证明了任意一个整数 `x` 的平方都是非负数，即：

```
sqr_pos: forall x: Z, x * x >= 0
```

我们可以借助这一性质完成上面 `sum_of_sqr2` 的证明。

```
Fact sum_of_sqr2: forall x y: Z,
  x * x + y * y >= 2 * x * y.
Proof.
  intros.
  pose proof sqr_pos (x - y).
  nia.
Qed.
```

这段证明有三个证明步骤。证明指令 `intros` 将待证明结论中的逻辑结构“对于任意整数 x 与 y”移除，并在前提中假如“x 与 y 是整数”这两条假设。第二条指令 `pose proof` 表示对 x-y 使用标准库中的定理

`sqr_pos` 。从 Coq 定理证明界面中可以看到，使用该定理得到的命题会被添加到证明目标的前提中去，Coq 系统将这个新命题自动命名为 H（表示 Hypothesis）。最后，`nia` 可以自动根据当前证明目标中的前提证明结论。

下面证明演示了如何使用我们刚刚证明的性质 `sum_of_sqr1` 。

```
Example quad_ex1: forall x y: Z,
  x * x + 2 * x * y + y * y + x + y + 1 >= 0.
Proof.
  intros.
  pose proof sum_of_sqr1 (x + y) (-1).
  nia.
Qed.
```

下面这个例子说的是：如果 `x < y` ，那么 `x * x + x * y + y * y` 一定大于零。

```
Fact sum_of_sqr_lt: forall x y: Z,
  x < y ->
  x * x + x * y + y * y > 0.
```

在我们学习数学知识时，我们知道 `x * x + x * y + y * y` 是恒为非负的，而且只有在 `x` 与 `y` 都为 0 的时候，这个式子才能取到 0，因此，在 `x < y` 的前提下，这个式子恒为正。在进一步学习 Coq 定理证明器的使用之后，我们完全可以在 Coq 中表达上述证明过程。不过，如果仅仅使用目前所介绍的几条 Coq 证明指令，其实也是可以完成上面命题的 Coq 证明的。不过，此处就需要换一条证明思路。

我们可以利用下面两式相等证明：

`4 * (x * x + x * y + y * y)`

`3 * (x + y) * (x + y) + (x - y) * (x - y)`

于是，在 `x < y` 的假设下，等式右边的两个平方式一个恒为非负，一个恒为正。因此，等式的左边也恒为正。将这一思路写成 Coq 证明如下：

```
Proof.
  intros.
  pose proof sqr_pos (x + y).
  nia.
Qed.
```

可以看到，在 `x < y` 的前提下，Coq 的 `nia` 指令可以自动推断得知 `(x - y)` 的平方恒为正。不过，我们仍然需要手动提示 Coq，`(x + y)` 的平方恒为非负。

Coq 证明脚本 2. intros 指令. 证明指令 `intros` 表示将待证明结论中的假设移动到证明目标的前提中去。例如，在上面 `sum_of_sqr_lt` 中，`intros` 指令移动了三项前提：`x: Z` 、`y: Z` 与 `H: x < y` 。其中 `H` 是 Coq 定理证明系统自动引入的命名，字母 H 表示 Hypothesis 的简写，当 `intros` 要添加若干个命题作为前提的时候，Coq 会依次选择 `H` 、`H0` 、`H1` 等名字。有时，我们在 Coq 中编写证明脚本代码时，希望能够手动控制这些前提的命名，这只需要在 `intros` 后添加参数就可以了。例如，`sum_of_sqr_lt` 中的 `intros` 指令就等效于 `intros x y H` 。Coq 允许我们在 `intros` 的同时对 `forall` 后的变量重命名，例如，将 `sum_of_sqr_lt` 中的 `intros` 指令改为 `intros x1 x2 H` 后效果如下。Coq 也允许对一部分前提手动命名，而同时对另一部分前提自动命名，只需用问号占位符表示自动命名的前提即可，例如 `intros ? ? H` 。

Coq 证明脚本 3. pose proof 指令. 证明指令 `pose proof` 表示在当前证明中使用一条已经证明过定理或者使用当前证明目标中的一条前提。例如，标准库中已有定理 `sqr_pos`

`sqr_pos: forall x: Z, x * x >= 0`

那么 `pose proof sqr_pos (x + 1)` 就会得到 `(x + 1) * (x + 1) >= 0` 。类似的，假设当前证明目标中有下述前提，

```
x: Z
H: x >= 0
H0: x >= 0 -> x + 1 > 0
```

那么，就可以通过 `pose proof H H0` 得到 `x + 1 > 0` 。另外，使用 `pose proof` 指令时未必需要将所有的前提全部填上，如 Coq 标准库中的 `Zmult_ge_compat_r` 是下面定理：

```
forall n m p : Z, n >= m -> p >= 0 -> n * p >= m * p
```

假设当前证明目标中有下述前提，

```
k1: Z
k2: Z
x: Z
H: k1 >= k2
H0: x * x >= 0
```

那么，就可以通过以下 `pose proof` 指令

```
pose proof Zmult_ge_compat_r k1 k2 (x * x) H
pose proof Zmult_ge_compat_r k1 k2 (x * x) H H0
pose proof Zmult_ge_compat_r (x * x) 0 5 H0 ltac:(lia)
```

分别得到以下结论：

```
x * x >= 0 -> k1 * (x * x) >= k2 * (x * x)
k1 * (x * x) >= k2 * (x * x)
x * x * 5 >= 0 * 5
```

可以看到，填写前提中的命题部分时，既可以填写已有前提的名称（如 `H` 、`H0` 等），也可以填写一条证明指令，如 `ltac:(lia)` 。除此之外，如果 `pose proof` 指令的一些参数可以由另一些参数推导出来，那么可以用下划线省去这些参数。例如，下面这几条证明指令的效果和上面证明指令的效果时相同的。

```
pose proof Zmult_ge_compat_r _ _ (x * x) H
pose proof Zmult_ge_compat_r _ _ _ H H0
pose proof Zmult_ge_compat_r _ _ 5 H0 ltac:(lia)
```

最后，在 Coq 中还可以指明 `pose proof` 所生成新命题的名称。例如，

```
pose proof Zmult_ge_compat_r _ _ 5 H0 ltac:(lia) as H5xx
```

得到的新命题是：`H5xx: x * x * 5 >= 0 * 5` 。

Coq 证明脚本 4. nia 指令. 证明指令 `nia` 表示自动证明有关整数非线性算数运算的性质，nia 这三个字母是 nonlinear integer arithmetic 的缩写。证明指令 `nia` 是不完备的，但是它能够自动完成多项式的展开与线性性质的推理。另外，它也能自动推到乘法与正负数之间的关系。

习题 2. 请证明下面结论。提示：可以利用以下代数变换完成证明

```
4 * (x * x + 3 * x + 4) = (2 * x + 3) * (2 * x + 3) + 7
```

```
Example quad_ex2: forall x: Z,
  x * x + 3 * x + 4 > 0.
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

2 函数与谓词

函数是一类重要的数学对象。例如，“加一”这个函数往往可以写作：$f(x) = x + 1$。在 Coq 中，我们可以用以下代码将其定义为 `plus_one` 函数。

```
Definition plus_one (x: Z): Z := x + 1.
```

在类型方面，`plus_one (x: Z): Z` 表示这个函数的自变量和值都是整数，而 `:=` 符号右侧的表达式 `x + 1` 也描述了函数值的计算方法。

我们知道，“在 1 的基础上加一”结果是 2，“在 1 的基础上加一再加一”结果是 3。这些简单论断都可以用 Coq 命题表达出来并在 Coq 中证明。

```
Example One_plus_one: plus_one 1 = 2.
Proof. unfold plus_one. lia. Qed.
```

```
Example One_plus_one_plus_one: plus_one (plus_one 1) = 3.
Proof. unfold plus_one. lia. Qed.
```

Coq 表达式 3. 包含函数的表达式. 在 Coq 中，某函数 `F` 作用于某参数 `X` 写作 `F X`，不需要写括号。这一语法类似于 Ocaml 等函数式编程语言。另外，这一语法是左结合的。换言之，表达式 `F X Y` 是 `(F X) Y` 的简写，而表达 `F (G X)` 时必须添加括号。

Coq 证明脚本 5. unfold 指令. 证明指令 `unfold` 表示在待证明的结论中展开某项定义。如果要在证明目标的某前提 `H` 中展开 `X` 的定义，可以使用证明指令 `unfold X in H` 。

下面是更多函数的例子。我们可以采用类似于定义“加一”的方法定义“平方”函数。

```
Definition square (x: Z): Z := x * x.
```

```
Example square_5: square 5 = 25.
Proof. unfold square. lia. Qed.
```

Coq 中也可以定义多元函数。

```
Definition smul (x y: Z): Z := x * y + x + y.
```

```
Example smul_ex1: smul 1 1 = 3.
Proof. unfold smul. lia. Qed.
```

```
Example smul_ex2: smul 2 3 = 11.
Proof. unfold smul. lia. Qed.
```

Coq 表达式 4. 包含多元函数的表达式. Coq 中的二元函数实质上是接收一个参数后会计算得到一个一元函数的函数。例如，当 `F` 是一个二元函数时，我们通常将“`F` 作用于 `X` 与 `Y` 的结果”写作 `F X Y`，即 `(F X) Y` 的简写。这是因为 `F X` 实质上是一个一元函数，当他再接收一个参数 `Y` 之后的计算结果就是 `(F X) Y` 。类似的，Coq 中的三元函数实质上是接收一个参数后会计算得到一个二元函数的函数；Coq 中的 n+1 元函数实质上是接收一个参数后会计算得到一个 n 元函数的函数。

下面 Coq 代码定义了“非负”这一概念。在 Coq 中，可以通过定义类型为 `Prop` 的函数来定义谓词。以下面定义为例，对于每个整数 `x`，`:=` 符号右侧表达式 `x >= 0` 是真还是假决定了 `x` 是否满足性质 `nonneg`（即，非负）。

```
Definition nonneg (x: Z): Prop := x >= 0.
```

```
Fact nonneg_plus_one: forall x: Z,
  nonneg x -> nonneg (plus_one x).
Proof. unfold nonneg, plus_one. lia. Qed.
```

```
Fact nonneg_square: forall x: Z,
  nonneg (square x).
Proof. unfold nonneg, square. nia. Qed.
```

习题 3. 请在 Coq 中证明下面结论。

```
Fact nonneg_smul: forall x y: Z,
  nonneg x -> nonneg y -> nonneg (smul x y).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 4. 我们可以使用乘法运算定义“正负号不同”这个二元谓词。请基于这一定义完成相关性质的 Coq 证明。

```
Definition opposite_sgn (x y: Z): Prop := x * y < 0.
```

```
Fact opposite_sgn_plus_2: forall x,
  opposite_sgn (x + 2) x ->
  x = -1.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Fact opposite_sgn_odds: forall x,
  opposite_sgn (square x) x ->
  x < 0.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 5. 下面定义的谓词 `quad_nonneg a b c` 表示：以 `a`、`b`、`c` 为系数的二次式在自变量去一切整数的时候都恒为非负。请基于这一定义完成相关性质的 Coq 证明。

```
Definition quad_nonneg (a b c: Z): Prop :=
  forall x: Z, a * x * x + b * x + c >= 0.
```

```
Lemma scale_quad_nonneg: forall a b c k: Z,
  k > 0 ->
  quad_nonneg a b c ->
  quad_nonneg (k * a) (k * b) (k * c).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Lemma descale_quad_nonneg: forall a b c k: Z,
  k > 0 ->
  quad_nonneg (k * a) (k * b) (k * c) ->
  quad_nonneg a b c.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Lemma plus_quad_nonneg: forall a1 b1 c1 a2 b2 c2: Z,
  quad_nonneg a1 b1 c1 ->
  quad_nonneg a2 b2 c2 ->
  quad_nonneg (a1 + a2) (b1 + b2) (c1 + c2).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

我们知道，如果二次式的二次项系数为正且判别式不为正，那么这个二次式在自变量取遍一切实数的时候都恒为正。相应的性质在自变量取遍一切整数的时候自然也成立。请证明这一结论。【选做】

```
Lemma plus_quad_discriminant: forall a b c,
  a > 0 ->
  b * b - 4 * a * c <= 0 ->
  quad_nonneg a b c.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

然而，判别式不为正并不是 `quad_nonneg` 的必要条件。下面命题是一个简单的反例。

```
Example quad_nonneg_1_1_0: quad_nonneg 1 1 0.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

3 逻辑连接词与逻辑命题

数学中可以用“并且”、“或”、“非”、“如果-那么”、“存在”以及“任意”把简单性质组合起来构成复杂性质或复杂命题，例如“单调且连续”与“无限且不循环”这两个常用数学概念的定义中就用到了逻辑连接词“并且”，又例如“有零点”这一数学概念的定义就要用到“存在”这个逻辑中的量词（quantifier）。Coq 中也允许用户使用这些常用的逻辑符号。

Coq 表达式 5. 逻辑表达式. Coq 标准库中定义的逻辑符号有：

- “并且”：`/\`
- “或”：`\/`
- “非”：`~`
- “如果-那么”：`->`
- “当且仅当”：`<->`
- “真”：`True`
- “假”：`False`
- “存在”：`exists`
- “任意”：`forall` 。

这些符号中，“存在”与“任意”的优先级最低，之后优先级从低到高依次是“当且仅当”、“如果-那么”、“或”、“并且”与“非”。值得一提的是，Coq 的二元逻辑连接词中“并且”、“或”以及“如果-那么”都是右结合的，换言之，`P /\ Q /\ R` 是 `P /\ (Q /\ R)` 的简写。Coq 中只有“当且仅当”这个逻辑连接词是左结合的。

下面是一个使用逻辑符号定义复合命题的例子。前面我们已经定义过 `nonneg` 表示一个整数非负，Coq 标准库中还提供了谓词 `Z.Odd` 和 `Z.Even` 表示一个整数是奇数/偶数。因此，我们就可以用 `nonneg n /\ Z.Even n` 表示整数 `n` 是一个非负偶数。

下面性质说的是，如果一个函数 `f` 能保持非负性，也能保持奇偶性，那么它也能保持“非负偶数”这个性质。

```
Fact logic_ex1: forall f: Z -> Z,
  (forall n, nonneg n -> nonneg (f n)) ->
  (forall n, Z.Odd n -> Z.Odd (f n)) ->
  (forall n, Z.Even n -> Z.Even (f n)) ->
  (forall n, nonneg n /\ Z.Even n -> nonneg (f n) /\ Z.Even (f n)).
```

不难发现，这一性质的证明“非负”的定义无关，也和奇偶性的定义无关，这一性质的证明只需用到其中各个逻辑符号的性质。下面是 Coq 证明。

```
Proof.
  intros.
  (** 在_[intros]_指令执行后，Coq证明目标中共有四条前提：
      - H: forall n: Z, nonneg n -> nonneg (f n)
      - H0: forall n: Z, Z.Odd n -> Z.Odd (f n)
      - H1: forall n: Z, Z.Even n -> Z.Even (f n)
      - H2: nonneg n /\ Z.Even n *)
  pose proof H n.
  (** 获得一个新的前提
      - H3: nonneg n -> nonneg (f n) *)
  pose proof H1 n.
  (** 获得一个新的前提
      - H3: Z.Even n -> Z.Even (f n) *)
  (** 而_[H0]_这个关于奇数的前提没有用 *)
  tauto.
Qed.
```

最后一条证明指令 `tauto` 是英文单词“tautology”的缩写，表示当前证明目标是一个命题逻辑永真式，可以自动证明。在上面证明中，如果把命题 `nonneg n` 与 `Z.Even n` 记作命题 `P1` 与 `Q1`，将命题 `nonneg (f n)` 看作一个整体记作 `P2`，将命题 `Z.Even (f n)` 也看作一个整体记为 `Q2`，那么证明指令 `tauto` 在此处证明的结论就可以概括为：如果

- `P1` 成立并且 `Q1` 成立（前提 `H1`）
- `P1` 能推出 `P2`（前提 `H2`）
- `Q1` 能推出 `Q2`（前提 `H3`）

那么 `P2` 成立并且 `Q2` 成立。不难看出，无论 `P1`、`Q1`、`P2` 与 `Q2` 这四个命题中的每一个是真是假，上述推导都成立。因此，这一推导过程可以用一个命题逻辑永真式刻画，`tauto` 能够自动完成它的证明。Coq 中也可以把这一原理单独地表述出来：

```
Fact logic_ex2: forall P1 Q1 P2 Q2: Prop,
  P1 /\ Q1 ->
  (P1 -> P2) ->
  (Q1 -> Q2) ->
  P2 /\ Q2.
Proof.
  intros P1 Q1 P2 Q2 H1 H2 H3.
  tauto.
Qed.
```

仅仅利用 `tauto` 指令就已经能够证明不少关于逻辑的结论了。下面两个例子刻画了一个命题与其逆否命题的关系。

```
Fact logic_ex3: forall {A: Type} (P Q: A -> Prop),
  (forall a: A, P a -> Q a) ->
  (forall a: A, ~ Q a -> ~ P a).
Proof. intros A P Q H a. pose proof H a. tauto. Qed.
```

```
Fact logic_ex4: forall {A: Type} (P Q: A -> Prop),
  (forall a: A, ~ Q a -> ~ P a) ->
  (forall a: A, P a -> Q a).
Proof. intros A P Q H a. pose proof H a. tauto. Qed.
```

Coq 证明脚本 6. tauto 指令. 如果当前证明目标可以完全通过命题逻辑永真式完成证明，那么 `tauto` 就可以自动构造这样的证明。这里，所谓完全通过命题逻辑永真式完成证明，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。另外，`tauto` 也能支持关于等式的简单证明，具体而言，`tauto` 会将形如 `a = a` 的命题看做“真”，将形如 `a <> a` 的命题看作“假”。

习题 6. 请在 Coq 中证明下面结论。

```
Fact logic_ex5: forall {A: Type} (P Q: A -> Prop),
  (forall a: A, P a -> Q a) ->
  (forall a: A, P a) ->
  (forall a: A, Q a).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 7. 请在 Coq 中证明下面结论。

```
Fact logic_ex6: forall {A: Type} (P Q: A -> Prop) (a0: A),
  P a0 ->
  (forall a: A, P a -> Q a) ->
  Q a0.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 8. 请在 Coq 中证明下面结论。

```
Fact logic_ex7: forall {A: Type} (P Q: A -> Prop) (a0: A),
  (forall a: A, P a -> Q a -> False) ->
  Q a0 ->
  ~ P a0.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 9. 请在 Coq 中证明下面结论。

```
Fact logic_ex8: forall {A B: Type} (P Q: A -> B -> Prop),
  (forall (a: A) (b: B), P a b -> Q a b) ->
  (forall (a: A) (b: B), ~ P a b \/ Q a b).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 10. 请在 Coq 中证明下面结论。

```
Fact logic_ex9: forall {A B: Type} (P Q: A -> B -> Prop),
  (forall (a: A) (b: B), ~ P a b \/ Q a b) ->
  (forall (a: A) (b: B), P a b -> Q a b).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```