简单 Coq 证明与 Coq 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 高阶函数

在 Coq 中，函数的参数也可以是函数。下面定义的 `shift_left1` 是一个二元函数，它的第一个参数 `f: Z -> Z` 是一个从整数到整数的一元函数，第二个参数 `x: Z` 是一个整数，这个函数依据两个参数计算出的函数值是 `f (x + 1)` 。

```
Definition shift_left1 (f: Z -> Z) (x: Z): Z :=
  f (x + 1).
```

下面是两个 `shift_left1` 的例子。

```
Example shift_left1_square: forall x,
  shift_left1 square x = (x + 1) * (x + 1).
Proof. unfold shift_left1, square. lia. Qed.
```

```
Example shift_left1_plus_one: forall x,
  shift_left1 plus_one x = x + 2.
Proof. unfold shift_left1, plus_one. lia. Qed.
```

从这两个例子可以看出，`shift_left1` 作为一个二元函数，也可以被看作一个一元函数，这个一元函数的参数是一个整数到整数的函数，这个一元函数的计算结果也是一个整数到整数的函数。例如，`shift_left1 square` 的计算结果在数学上可以写作函数

$$f(x) = (x+1)^2。$$

更一般的，`shift_left1 f` 可以看作将函数 `f` 在坐标平面中的图像左移一个单位的结果。这样的观点也可以在 Coq 中更直白地表达出来，下面定义的 `shift_left1'` 实际上是与 `shift_left1` 完全相同的函数，但是这一定义在形式上是一个一元函数。

```
Definition shift_left1' (f: Z -> Z): Z -> Z :=
  fun x => f (x + 1).
```

Coq 表达式 5. 匿名函数与保留字 `fun`. Coq 中可以使用保留字 `fun` 表示匿名函数。例如，`fun x: Z => x + 10` 表示这样的一个匿名函数，它接收一个整数参数，如果这个参数的值为 `x`，那么这个函数的计算结果是 `x + 10`。当参数的类型可以自动推断得出时，上面参数的类型也可以省略。匿名函数也可以是多元函数，例如 `fun (f: Z -> Z) (x: Z) => f (x + 1)`。Coq 中的匿名函数语法取自于著名的 lambda 表达式。

与 `shift_left1` 类似，我们还可以定义将一元函数图像向上移动一个单位的结果。

```
Definition shift_up1 (f: Z -> Z) (x: Z): Z :=
  f x + 1.
```

```
Example shift_up1_square: forall x,
  shift_up1 square x = x * x + 1.
Proof. unfold shift_up1, square. lia. Qed.
```

```
Example shift_up1_plus_one: forall x,
  shift_up1 plus_one x = x + 2.
Proof. unfold shift_up1, plus_one. lia. Qed.
```

像 `shift_left1` 和 `shift_up1` 这样以函数为参数的函数称为高阶函数。高阶函数也可以是多元函数。例如下面的 `func_plus` 与 `func_mult` 定义了函数的加法和乘法。

```
Definition func_plus (f g: Z -> Z): Z -> Z :=
  fun x => f x + g x.
```

```
Definition func_mult (f g: Z -> Z): Z -> Z :=
  fun x => f x * g x.
```

下面证明几条关于高阶函数的简单性质。首先我们证明，对于任意函数 `f`，对它先左移再上移和先上移再左移的结果是一样的。

```
Lemma shift_up1_shift_left1_comm: forall f,
  shift_up1 (shift_left1 f) = shift_left1 (shift_up1 f).
Proof.
  intros.
  unfold shift_left1, shift_up1.
  reflexivity.
Qed.
```

在上面证明中，在展开“左移”与“上移”的定义后，需要证明的结论是：

```
(fun x : Z => f (x + 1) + 1) = (fun x : Z => f (x + 1) + 1)
```

这个等式两边的表达式字面上完全相同，因此该结论显然成立。证明指令 `reflexivity` 表示利用“自反性”完成证明，所谓等式的自反性就是“任意一个数学对象都等于它自身”。下面我们还可以证明，将 `f` 与 `g` 两个函数先相加再图像左移，与先图像左移再函数相加得到的结果是一样的。

```
Lemma shift_left1_func_plus: forall f g,
  shift_left1 (func_plus f g) =
  func_plus (shift_left1 f) (shift_left1 g).
Proof.
  intros.
  unfold shift_left1, func_plus.
  reflexivity.
Qed.
```

习题 6. 请证明下面命题。

```
Fact shift_up1_eq: forall f,
  shift_up1 f = func_plus f (fun x => 1).
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

4 高阶谓词

类似于高阶函数，如果一个谓词的参数中有函数（或谓词），那它就是一个高阶谓词。其实，许多常见数学概念都是高阶谓词。例如，函数的单调性就是一个高阶谓词。

```
Definition mono (f: Z -> Z): Prop :=
  forall n m, n <= m -> f n <= f m.
```

有许多函数都是单调的。前面我们定义的 `plus_one` 函数就是个单调函数。

```
Example plus_one_mono: mono plus_one.
Proof.
  unfold mono, plus_one.
  intros.
  lia.
Qed.
```

我们还可以定义整数函数的复合，然后证明复合函数能保持单调性。

```
Definition Zcomp (f g: Z -> Z): Z -> Z :=
  fun x => f (g x).
```

```
Lemma mono_compose: forall f g,
  mono f ->
  mono g ->
  mono (Zcomp f g).
Proof.
  unfold mono, Zcomp.
  intros f g Hf Hg n m Hnm.
  pose proof Hg n m Hnm as Hgnm.
  pose proof Hf (g n) (g m) Hgnm.
  lia.
Qed.
```

习题 7. 请证明常值函数都是单调的。

```
Lemma const_mono: forall a: Z,
  mono (fun x => a).
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 8. 请证明立方函数是单调的。

```
Example cube_mono: mono (fun x => x * x * x).
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 9. 请证明函数加法能保持单调性。

```
Lemma mono_func_plus: forall f g,
  mono f ->
  mono g ->
  mono (func_plus f g).
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

“结合律”也是一个常用的数学概念。如下面定义所示，一个二元函数 `f` 具有结合律，当且仅当它满足 `f x (f y z) = f (f x y) z` 。

```
Definition assoc (f: Z -> Z -> Z): Prop :=
  forall x y z,
    f x (f y z) = f (f x y) z.
```

我们熟知整数的加法于乘法都满足结合律。下面是加法结合律与乘法结合律的 Coq 证明。

```
Lemma plus_assoc: assoc (fun x y => x + y).
Proof. unfold assoc. lia. Qed.
```

```
Lemma mult_assoc: assoc (fun x y => x * y).
Proof. unfold assoc. nia. Qed.
```

习题 10. 请证明，我们先前定义的 `smul` 函数也符合结合律。

```
Lemma smul_assoc: assoc smul.
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

上面例子中的两个高阶谓词都是以函数为参数的高阶谓词，下面两个例子都是以谓词为参数的高阶谓词，甚至它们的参数本身也是高阶谓词。它们分别说的是，函数的性质 `P` 能被图像上移变换（`shift_up1` ）保持以及能被图像左移变换（`shift_left1` ）保持。

```
Definition preserved_by_shifting_up (P: (Z -> Z) -> Prop): Prop :=
  forall f, P f -> P (shift_up1 f).
```

```
Definition preserved_by_shifting_left (P: (Z -> Z) -> Prop): Prop :=
  forall f, P f -> P (shift_left1 f).
```

不难发现，单调性就能被这两种图像平移变换保持。

```
Lemma mono_pu: preserved_by_shifting_up mono.
(* 证明详见Coq源代码。 *)
```

```
Lemma mono_pl: preserved_by_shifting_left mono.
(* 证明详见Coq源代码。 *)
```

习题 11. 请证明“恒为非负”这一性质（见下面 Coq 定义）也被图像上移与图像左移变换保持。

```
Definition univ_nonneg {A: Type} (f: A -> Z): Prop :=
  forall a: A, f a >= 0.
```

```
Lemma univ_nonneg_pu: preserved_by_shifting_up univ_nonneg.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Lemma univ_nonneg_pl: preserved_by_shifting_left univ_nonneg.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

在数学证明中，我们往往会先证明一些具有一般性的定理或引理，并后续的证明中使用这些定理或引理。站在编写 Coq 证明脚本的角度看，使用先前已经证明的定理也可以看作对定理本身证明代码的复用。反过来，如果一系列数学对象具有相似的性质，这些性质的证明方式上也是相似的，那么我们在 Coq 编码时就可以利用函数与谓词（特别是高阶函数和高阶谓词）概括性地描述这些数学对象之间的关系，并给出统一的证明。例如，当我们要证明

$$f(x) = x^3 - 3x^2 + 3x$$

具备单调性时，可以基于下面代数变换并多次使用复合函数保持单调性这一引理完成证明。

$$f(x) = x^3 - 3x^2 + 3x = (x-1)^3 + 1$$

```
Example mono_ex1: mono (fun x => x * x * x - 3 * x * x + 3 * x).
```

具体而言，下面在 Coq 中证明这一单调性性质的时候，将分两步进行。第一步先证明 `fun x => x - 1`、`fun x => x + 1` 与 `fun x => x * x * x` 这三个简单函数都是单调的；第二步则利用复合函数的单调性性质将这三个结论组合起来完成证明。

```
Proof.
  assert (mono (fun x => x - 1)) as H_minus.
  (** 这里的_[assert]_指令表示：声明_[mono (fun x => x - 1)]_这一命题成立。逻辑上
      看，一个合理的逻辑系统不应当允许我们凭空声明一条性质。因此，我们在完成声明后需要
      首先证明“为什么这一命题成立”，再基于此进行后续证明。在执行完这一条指令后，Coq系
      统显示目前有两个证明目标需要证明，它们分别对应这里所说的“为什么这一命题成立”与后
      续证明两部分。*)
  { unfold mono. lia. }
  (** 在有多个证明目标需要证明的时候，证明脚本中的左大括号表示进入其中的第一个证明目标
      的证明。这里的第一个证明目标就是要证明_[fun x => x - 1]_是一个单调函数。该证明
      结束后，证明脚本中的右大括号表示返回其余证明目标。返回后可以看到，证明目标中增加
      了一条前提：_[H_minus: fun x => x - 1]_。这一前提的名称_[H_minus]_是先前的
      _[assert]_指令选定的。*)
  assert (mono (fun x => x + 1)) as H_plus.
  { unfold mono. lia. }
  (** 类似的，这里可以使用声明+证明的方式证明_[fun x => x + 1]_也是一个单调函数。*)
  pose proof cube_mono as H_cube.
  (** 先前已经证明过，立方函数是单调的，这里直接_[pose proof]_这一结论。至此，我们已
      经完成了三个子命题的证明，下面的将通过复合函数的形式把它们组合起来。*)
  pose proof mono_compose _ _ H_plus (mono_compose _ _ H_cube H_minus).
  unfold mono.
  intros n m Hnm.
  unfold mono, Zcomp, plus_one in H.
  pose proof H n m Hnm.
  nia.
Qed.
```

Coq 证明脚本 6. assert 指令. 如果 `P` 是一个 Coq 命题，那么 `assert(P)` 指令可以将当前证明目标规约为两个目标：其一是用当前的前提推导 `P` ；其二是在使用当前的前提与 `P` 共同推导当前的结论。如果要对新增的前提 `P` 手动命名，可以采用形如 `assert(P) as H99` 的指令；如果交由 Coq 系统自动命名，它的名称将是 `H` 、`H0` 、`H1` ... 中第一个可以使用的名字。

习题 12. 请证明下面 Coq 命题。

```
Example mono_ex2: mono (fun x => x * x * x + 3 * x * x + 3 * x).
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

在先前证明单调函数性质的过程中，我们经常会要通过参数之间的大小关系推导出函数值之间的大小关系。例如，当 f 是一个单调函数时，可以由 $x \leqslant y$ 推出 $f(x) \leqslant f(y)$。对于一般的函数，我们也可以由参数相等，推出函数值相等，即由 $x = y$ 推出 $f(x) = f(y)$。下面的证明中就要用到这一性质。

```
Definition is_fixpoint (f: Z -> Z) (x: Z): Prop :=
  f x = x.
```

```
Theorem fixpoint_self_comp: forall f x,
  is_fixpoint f x ->
  is_fixpoint (Zcomp f f) x.
Proof.
  unfold is_fixpoint, Zcomp.
  intros.
  (** 此时需要利用前提_[H: f x = x]_证明_[f (f x) = x]_。*)
  rewrite H.
  rewrite H.
  reflexivity.
Qed.
```

在数学上，如果 `f x = x` ，那么我们就称 `x` 是函数 `f` 的一个不动点。上面的定理证明了，如果 `x` 是 `f` 的不动点，那么 `x` 也是 `f` 与自身复合结果的不动点。在这一证明中，前提 `H` 是命题 `f x = x` 。证明指令 `rewrite H` 的效果是将结论中的 `f x` 替换为 `x` ，因此，第一次使用该指令后，原先需要证明的 `f (f x) = x` 规约为了 `f x = x` 。这一步背后的证明实际就用到了“只要函数 `f` 的参数不变，那么函数值也不变”这条性质。

Coq 证明脚本 7. 利用等式做证明的 rewrite 指令。 有时，这个 `a` 可能出现了多次，但是我们只希望将其中的若干个 `a` 而不是全部的 `a` 都替换成 `b` ，此时可以在 rewrite 指令中增加 `at` ，即使用 `rewrite H at ...` 或 `rewrite H at ... in ...` 指明需要进行替换的位置。例如，当前提 `H` 为 `x = f x` ，待证明结论为 `x = f (f x)` 时，

`rewrite H` 与 `rewrite H at 1, 2`

都会将结论变为 `f x = f (f (f x))` ；

`rewrite H at 1`

会将结论变为 `f x = f (f x)` ；而

`rewrite H at 2`

会将结论变为 `x = f (f (f x))` 。

Coq 允许 rewrite 使用的定理或前提中有 `forall` 概称量词，用户可以手动地填入这些 `forall` 约束的变量或由 Coq 自动填入这些变量。例如，当前提 `H1` 与待证明结论分别为：

```
H1: forall x y: Z, g x y = g y x
结论: g 3 5 = 6
```

时，`rewrite H1` 、`rewrite (H1 3)` 或 `rewrite (H1 3 5)` 都会将待证明结论变换为 `g 5 3 = 6` 。Coq 还允许 rewrite 使用的定理或前提中带有附加条件，例如当前提 `H2` 与待证明结论分别为：

```
H2: forall x: Z, x <= 0 -> h x = 0
结论: h (h (-5)) = 0
```

时，用户可以直接指明如何证明这一附加条件，使用 `rewrite (H2 (-5) ltac:(lia))` 完成重写替换；用户也可以不直接指明，例如 `rewrite H2` 或者 `rewrite (H2 (-5))` 会将原证明目标规约为两个新的证明目标：第一个证明目标中结论变为 `h 0 = 0` ，第二个证明目标为需要补充证明的附加条件 `-5 <= 0` 。如果使用的定理或前提带有超过一个附加条件，那么 `rewrite` 指令就会生成超过两个证明目标。如果产生的所有附加条件都可以用一条证明脚本完成证明，那么可以在 rewrite 指令中加入 `by` 。上面例子中，`rewrite H2 by lia` 可以直接将结论变为 `h 0 = 0` 并不再额外产生附加的证明条件。

最后，Coq 中不仅允许将等式左侧的内容替换为等式右侧的内容，也允许使用 `<-` 进行反向操作。例如，当 `H` 具有形式 `a = b` 时，`rewrite <- H` 会将结论中的 `b` 替换为 `a` 。同时，Coq 也允许在一条 rewrite 指令中，按指定顺序连续进行多次替换，例如 `rewrite H1, H2` 表示先 `rewrite H1` 再 `rewrite H2` 。

下面关于不动点简单性质的证明需要我们灵活使用 rewrite 指令。

```
Example fixpoint_self_comp23: forall f x,
  is_fixpoint (Zcomp f f) x ->
  is_fixpoint (Zcomp f (Zcomp f f)) x ->
  is_fixpoint f x.
Proof.
  unfold is_fixpoint, Zcomp.
  intros.
  rewrite H in H0.
  rewrite H0.
  reflexivity.
Qed.
```

习题 13. 请在 Coq 中证明下面关于结合律的性质。

```
Definition power2 (f: Z -> Z -> Z) (x: Z): Z :=
  f x x.
```

```
Definition power4 (f: Z -> Z -> Z) (x: Z): Z :=
  f x (f x (f x x)).
```

```
Fact power2_power2: forall f a,
  assoc f ->
  power2 f (power2 f a) = power4 f a.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 14. 请在 Coq 中证明下面关于群的性质。提示：`f x (g x) = f 1 (f x (g x))` 而 `f (g (g x)) (g x) = 1` 。

```
Fact group_basic: forall (f: Z -> Z -> Z) (g: Z -> Z),
  assoc f ->
  (forall x, f 1 x = x) ->
  (forall x, f (g x) x = 1) ->
  (forall x, f x (g x) = 1).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

5 类型多态

前面我们曾在 Coq 中定义了整数函数之间的复合 `Zcomp` ，然而，函数复合的概念不仅仅限于整数函数之间，例如，高阶函数 `shift_up1` 与 `shift_left1` 之间也可以复合。Coq 允许我们统一定义所有这些函数复合的概念。具体而言，只要 `f` 是一个 `B -> C` 类型的函数（参数类型为 `B` 计算结果类型为 `C` ），并且 `g` 是一个 `A -> B` 类型的函数，那么就可以将它们复合，得到一个类型为 `A -> C` 的函数。像 `comp` 这样以类型为参数的函数，有时也被称为类型多态函数，或者多态函数。

```
Definition comp {A B C: Type} (f: B -> C) (g: A -> B): A -> C :=
  fun x => f (g x).
```

在 `comp` 的定义中共有五个参数，其中 `A` 、`B` 与 `C` 这三个参数用大括号表示它们是隐式参数，即在使用函数 `comp` 不需要填充这三个参数。例如：

```
Example comp_ex1:
  comp square plus_one = fun x => (x + 1) * (x + 1).
Proof. unfold comp, square, plus_one. reflexivity. Qed.
```

Coq 也允许使用 `@` 符号将所有参数都变为显式参数。例如：

```
Example comp_ex1':
  @comp Z Z Z square plus_one = fun x => (x + 1) * (x + 1).
Proof. unfold comp, square, plus_one. reflexivity. Qed.
```

下面例子讨论了 `shift_up1` 与 `shift_left1` 之间复合的性质，它说的是无论 `shift_up1` 与 `shift_left1` 复合时哪个在左哪个在右，复合的结果是相同的。

```
Example comp_ex2:
  comp shift_left1 shift_up1 = comp shift_up1 shift_left1.
Proof. unfold comp, shift_left1, shift_up1. reflexivity. Qed.
```

由于类型多态的 `comp` 函数是比 `Zcomp` 更一般的函数，我们也可以证明更一般的性质。例如，下面我们可以证明函数复合的结合律。这里值得指出的是：这一定理中涉及四次函数复合，从左向右出现的四次函数复合分别是关于 `A-C-D` 类型、`A-B-C` 类型、`A-B-D` 类型与 `B-C-D` 类型的函数复合。

```
Theorem comp_assoc: forall (A B C D: Type) (f: C -> D) (g: B -> C) (h: A -> B),
  comp f (comp g h) = comp (comp f g) h.
Proof.
  intros.
  unfold comp.
  reflexivity.
Qed.
```

另一个常用的多态函数是 `swap` ，它会交换一个二元函数的两个参数。

```
Definition swap {A B C: Type}: (A -> B -> C) -> (B -> A -> C) :=
  fun f x y => f y x.
```

例如：

```
Example swap_ex1: swap (fun x y => x - y) = (fun x y => y - x).
Proof. unfold swap. reflexivity. Qed.
```

```
Example swap_ex2: swap Zcomp = (fun f g x => g (f x)).
Proof. unfold swap, Zcomp. reflexivity. Qed.
```

习题 15. 请在 Coq 中证明下面关于 `swap` 的性质。

```
Lemma swap_involutive: forall (A B C: Type) (f: A -> B -> C),
  swap (swap f) = f.
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 16. 下面定义的 `doit3times` 表示将函数 `f` 复合 3 次。

```
Definition doit3times {X: Type} (f: X -> X) (n: X): X :=
  f (f (f n)).
```

请回答下面几个关于 `doit3times` 的问题。

- `doit3times (fun x => x * x) 2` 的值是多少？
- `doit3times (fun x => - x) 5` 的值是多少？
- `doit3times swap (fun x y => x - y) 1 2` 的值是多少？
- `doit3times (func_plus (fun x => x - 2)) (fun x => x * x)` 相当于什么函数？
- `doit3times ((fun x y => y * y - x * y + x * x) 1) 1` 的值是多少？

习题 17. 请回答下面几个关于 `doit3times` 的问题。

- `doit3times (fun x => x - 2) 9` 的值是多少？
- `doit3times (fun x => x - 2) (doit3times (fun x => x - 2) 9)` 的值是多少？
- `doit3times (doit3times (fun x => x - 2)) 9` 的值是多少？
- `doit3times doit3times (fun x => x - 2) 9` 的值是多少？

习题 18. 请回答下面几个关于类型多态函数的问题。

- `doit3times shift_left1 (fun x => x) 0` 的值是多少？
- `doit3times shift_left1 (fun x => x - 2) 0` 的值是多少？
- `doit3times shift_left1 (fun x => x * x) 0` 的值是多少？
- `comp (fun x => x - 2) (fun x => x * x) 10` 的值是多少？
- `comp (fun x => x * x) (fun x => x - 2) 10` 的值是多少？
- `doit3times swap comp (fun x => x * x) (fun x => x - 2) 10` 的值是多少？