Coq 手册 1

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。

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。

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。

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` 。

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。

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。

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。

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` 。