Coq 手册 2

Coq 表达式 1. 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。

Coq 证明脚本 1. 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”。

Coq 证明脚本 2. apply 指令.

如果待证明结论与某个前提或者定理是相同的命题，那么就可以用 `apply` 指令完成证明。如果前提或定理中还包含额外的概称量词，`apply` 也能完成证明。例如，当前提 `H` 与结论分别为下面命题时

```
H: forall x: Z, x < x + 1
结论: u + 1 < (u + 1) + 1
```

`apply H` 或者 `apply (H (u + 1)` 都可以完成证明。除此之外，`apply` 指令还可以处理前提或者定理中包含一些附加条件的情况。例如，`Z.mul_nonneg_nonneg` 这条 Coq 标准库中的定理说的是:

```
forall n m, 0 <= n -> 0 <= m -> 0 <= n * m
```

于是，如果要证明 `0 <= 5 * n` 就可以使用下面的证明指令之一：

```
apply Z.mul_nonneg_nonneg
apply (Z.mul_nonneg_nonneg 5)
apply (Z.mul_nonneg_nonneg 5 n)
```

它们的效果都是将当前证明目标拆分两个：分别证明 `0 <= 5` 与 `0 <= n` 这两个结论。还可以使用下面证明指令：

```
apply (Z.mul_nonneg_nonneg 5 n ltac:(lia))
```

这样原结论就被规约为 `0 <= n` 这一个命题了。

Coq 证明脚本 3. left 指令与 right 指令.

如果待证明结论具有 `P \/ Q` 的形式，那么 `left` 可以将该结论规约为 `P` ， `right` 可以将该结论规约为 `Q` 。

Coq 证明脚本 4. 命题逻辑证明中的 split 指令.

如果待证明结论具有 `P /\ Q` 的形式，那么 `split` 可以将当前证明目标规约为两个更简单的证明目标，它们的前提与原证明目标的前提相同，它们的结论分别为 `P` 与 `Q` 。

Coq 证明脚本 5. 命题逻辑证明中的 destruct 指令.

如果证明时前提 `H` 具有形式 `P /\ Q` 或具有形式 `P \/ Q` ，则可以使用 `destruct H` 指令。当 `H` 具有形式 `P /\ Q` 时，该指令会将此前提分解为两个前提 `P` 与 `Q` 。当 `H` 具有形式 `P \/ Q` 时，该指令会将当前证明目标规约为两个证明目标，其中一个将前提 `H` 被改为 `P` ，另一个将前提 `H` 改为 `Q` 。

Coq 允许用户使用 `destruct ... as ...` 指令对 destruct 得到的新前提手动重命名。例如当 `H` 具有形式 `P /\ Q` 时， `destruct H as [H1 H2]` 指令将生成下面两个证明前提：

```
H1: P
H2: Q
```

又例如当 `H` 具有形式 `P \/ Q` 时， `destruct H as [H1 | H2]` 指令也可以用于手动命名。与 intros 指令中的规定一样，当需要对 destruct 结果中的一部分手动命名而对另一部分自动命名时，可以使用问号 `?` 表示那些需要由 Coq 自动命名的名字。

Coq 允许用户对多个前提同时执行 destruct 指令,例如 `destruct H, H0` 就表示先 `destruct H` 再 `destruct H0` 。Coq 也允许用户在一条 destruct 指令中对 destruct 的结果再进一步分解或进一步分类讨论，但具体需要分解多少层，需要使用 `destruct ... as ...` 指令做具体说明。例如，当 `H` 具有形式

```
(P /\ Q) /\ (R /\ S)
```

时， `destruct H as [? [? ?]]` 指令会将 `H` 分解为 `P /\ Q` 、 `R` 与 `S` ；而 `destruct H as [[? ?] ?]` 指令会将 `H` 分解为 `P` 、 `Q` 与 `R /\ S` 。另外，对“并且”与“或”的分解与分类讨论也可以相互嵌套，例如，当 `H` 具有形式

```
(P /\ Q) \/ R
```

时，可以使用 `destruct H as [[HP HQ] | HR]` 指令在分类讨论的同时对其中一个分类讨论的分支对前提做进一步分解。

Coq 证明脚本 6. 引入模式.

前面已经介绍， `destruct ... as ...` 指令可以一次性完成若干次的命题拆解或分类讨论。在这一指令中， `as` 之后的结构成为“引入模式”（intro-pattern）。以下是与目前所学相关的几种引入模式：

- 针对“并且”的命题拆解 `[intro_pattern1 intro_pattern2]` ；
- 针对“或”的分类讨论 `[intro_pattern1 | intro_pattern2]` ；
- 新引入的名字，例如 `H` ；
- 表示由 Coq 系统自动命名的问号 `?` ；
- 表示直接丢弃拆分结果的下划线 `_` ；

除了 destruct 指令之外，intros 指令与 pose proof 指令也可以使用引入模式，在完成原有功能的基础上，再进行一次 destruct。例如，`intros [H1 H2] [H3 | H3]` 相当于依次执行：

```
intros H1 H2
destruct H1 as [H1 H2]
destruct H3 as [H3 | H3] ;
```

而 `pose proof H x y as [H1 H2]` 相当于依次执行：

```
pose proof H x y as H1
destruct H1 as [H1 H2] 。
```