第 3 章 命题和证明

本章，我们将基于一个非常精简的逻辑，**最小命题逻辑**来介绍 *Coq* 中的推理技术。在最小命题逻辑中，公式由命题变量和蕴含式构造而成。例如，P、Q、R 为命题，让我们考虑如何证明如下公式：

$$(P \Rightarrow Q) \Rightarrow ((Q \Rightarrow R) \Rightarrow (P \Rightarrow R))$$

有两种方法可以解决这个问题。

第一种方法由 Tarski [80] 提出。我们可以对每一个变量进行赋**值**：***t***（true）或 ***f***（false）。如果对于所有的赋值，公式的结果均为真，那么公式为真。我们可以使用真值表来验证公式是否成立。该表枚举所有可能的赋值。如果对于表中的每一行，计算结果均为真，则公式是**成立的**。上面的公式是成立的，其真值表如图 3.1 所示。

P	Q	R	P⇒Q	Q⇒R	P⇒R	(Q⇒R)⇒(P⇒R)	(P⇒Q)⇒(Q⇒R)⇒(P⇒R)
f	*f*	*f*	*t*	*t*	*t*	*t*	*t*
f	*f*	*t*	*t*	*t*	*t*	*t*	*t*
f	*t*	*f*	*t*	*f*	*t*	*t*	*t*
f	*t*	*t*	*t*	*t*	*t*	*t*	*t*
t	*f*	*f*	*f*	*t*	*f*	*f*	*t*
t	*f*	*t*	*f*	*t*	*t*	*t*	*t*
t	*t*	*f*	*t*	*f*	*f*	*t*	*t*
t	*t*	*t*	*t*	*t*	*t*	*t*	*t*

图 3.1 真值表

第二种方法由 Heyting [50]提出。该方法将问题“命题 P 是否为真？”转化为“如果命题 P 有证明，那么这些证明是什么？” *Coq* 采纳了这种方法。为此，我们不但需要表达公式，还需要表达公式的证明。事实上，我们可以使用在前一章学习的带类型的 λ-演算对证明进行表达。依照 Heyting 的方法，公式 $P \Rightarrow Q$ 的证明就是从 P 的证明得到 Q 的证明的一个过程。换言之，$P \Rightarrow Q$ 的证明就是一个函数，该函数的输入为任意的 P 的证明，输出为 Q 的证明。

上面提到的两种方法代表了对逻辑的两种不同认识。Tarski 的方法对应于经典逻辑，在此逻辑中排中律“对于所有的命题，非真即假”是成立的。这点对应于真值表的

使用。Heyting 的方法对应于直觉逻辑，在此逻辑中排中律是不成立的。在 3.7.2 节中我们将看到，在经典逻辑中成立的一些命题并不能在直觉逻辑中被证明。

虽然直觉逻辑的能力相对于经典逻辑较弱，但是对于计算机科学家来说，它具有一个很好的性质：我们可以从证明中抽取正确的程序。这也是 *Coq* 之所以采纳 Heyting 方法的主要原因。

一旦我们采纳了 Heyting 方法，采用上一章中提到的函数式程序设计技术将变得非常自然。如果我们将证明看成函数式语言中的表达式，那么待证明的命题则可以被视为一个类型（证明的类型）。Heyting 的蕴含式 $P \Rightarrow Q$ 可以看成箭头类型 $P{\to}Q$。蕴含式的证明可以看成一个形如“`fun H:`$P \Rightarrow t$”简单的抽象，它表示，在 P 的假设 H 下，t 为 Q 的一个合式证明。

作为函数式程序设计模型的 λ-演算与像自然推理系统 [77] 这样的证明演算（proof calculi）之间的对应关系被称为“Curry–Howard 同构”。有很多科学家在此问题上进行了深入的研究：Scott [79]，Martin-Löf [63]，Girard，Lafont，和 Taylor [47]和 Feys [33]和 Howard [53]。正因为有这种对应关系，使得我们在证明时可以使用程序设计的观点，在程序设计时使用逻辑的观点。此外，*Coq* 里的工具不但可以应用在程序设计方面，亦可以应用在推理方面。

为了进一步统一程序设计技术和推理技术，我们将放弃仅在逻辑中使用的蕴含符号，用记号 $P{\to}Q$ 表示来表示“P 蕴含 Q”。例如，“蕴含 Q”。前面我们提到的命题将写为下面的形式：

$$(P{\to}Q){\to}(Q{\to}R){\to}P{\to}R$$

这个命题的证明是一个类型为该命题的 λ-项，比如：

`fun (H:`$P{\to}Q$`)(H':`$Q{\to}R$`)(p:`P`) ⇒ H' (H p)`

这个项表达了在给定 `P→Q`，`Q→R`，和 `P` 的任意证明时，如何构造 `R` 的证明（其中，`H` 为 `P→Q` 的证明，`H'` 为 `Q→R` 的证明，`p` 为 `P` 的证明）。该构造过程如下：通过将 `H` 作用到 `p` 上得到 `Q` 的证明，然后将 `H'` 作用到 `Q` 的证明上得到 `R` 的证明。这种推理方式就是著名的**三段论法**（syllogism）。

在本章中，我们将对上一章介绍的类型系统进行修改，使其能同时支持证明和程序。我们还将介绍一些能让定理证明过程变得简单的工具。在这些工具中，**证明策略**发挥了重要的作用。这些证明策略使得我们可以半自动化地得到证明。构造证明和构造程序的过程是类似的，但需要注意的是：在进行证明时，我们只关心是否能找到一个证明，而不关心证明的复杂程度。另一方面，并不是所有满足同一规范说明的程序都类似，即使它们的最终结果是一样的。程序的效率是我们必须考虑的问题。证明的细节无关紧要这一观点——通常被称为**证明无关性**——使得我们可以放心地让计算机来寻找证明。但是，因为效率的原因，我们不愿让计算机为我们设计程序。

3.1 最小命题逻辑

在最小命题逻辑中，我们考虑仅由命题变量和蕴含关系构成的逻辑公式。由于我们用箭头表示蕴含关系，所以这里的逻辑命题和前一章的规范说明在结构上完全相同。

3.1.1 命题和证明

一方面是程序和证明，另一方面是说明和命题，我们可以用一个新的大类来实现两者的共存: 它就是表示命题的 Prop 大类，在类型的层次结构中该大类与 Set 大类扮演着相似的角色。对任意的 i 该大类的类型是 Type(i)，我们可以用下面的公式来表示：

$$E, \Gamma \vdash \texttt{Prop} \leq_{\delta\beta\zeta\iota} \texttt{Type(i)} \text{ 对每一个} i \text{ 都成立}$$

Prop 大类可以用于定义命题和证明，就像 Set 大类可用于规范说明和程序那样。

定义 4 **命题，证明项**： 若类型 P 的类型为大类 Prop，则它被称为一个**命题**。若项 t 的类型为命题，则它被称为一个**证明项**，或简称为**证明**。

一种经常出现的情况是，一个命题只能在一组给定的命题假设之下才能证明。我们可以像在程序中声明变量那样来声明这些假设命题（参见 2.2.2 节）。事实上，假设和局部声明是相同的，就像公理和全局声明是相同的一样。

定义 5 **假设**： 若 h 是一个标识符，P 是一个命题，则局部声明 $h : P$ 被称为一个**假设**。标识符 h 是假设**名**，P 是假设**命题**。

我们推荐使用命令“Hypothesis h:P”来声明一个假设，尽管该命令与“Variable h:P” 同义。若要同时声明多个假设，可以使用关键字 Hypotheses（和 Variables 同义）。

假设的作用是将 h 声明为 P 的任意一个证明项。这个声明在局部范围内有效，这使得我们仅可以使用这个假设到当前 section 结束。使用假设是**自然演绎** [77] 的核心推理方法。

定义 6 **公理**： 若 x 是一个标识符，P 是一个命题，则全局声明 $x : P$ 被称为一个**公理**。

命令“Axiom x:P” 和“Parameter x:P” 是同义的，但当 P 为命题时，应优先使用前者。

公理不同于假设，公理对其后的证明部分总是成立的，即使在当前 section 之外。声明公理 $x : P$ 和假设 P 为真是一样的。在证明中使用公理是有危险的，因为人们可能忘记证明是建立在这些公理成立的基础之上的。在练习 6.13 中，我们给出一个由不适宜的公理而导致矛盾的例子。

回顾第 2 章中介绍的表示方法，我们知道：环境包括了证明一个定理所需要的所有公理，而上下文包含了所有当前假设。判定符号

$$E, \Gamma \vdash \pi : P$$

可以读作：

在 E 中的公理和 Γ 中的假设下，π 是 P 的一个证明。

定义 7 **定理，引理：** 若标识符的全局定义的类型是一个命题，则该全局定义被称为**定理**或**引理**。

再次回顾上一章中的表示方法，我们会得知：环境 E 和上下文 Γ 中既包含了当前证明的假设条件，也包含了当前证明的已证事实。

3.1.2 目标和证明策略

对于简单的命题，证明过程也可能很复杂。为此，*Coq* 系统提供了一套辅助工具。该工具的工作模式是用户将要证明的命题作为**目标**（goal），然后使用**证明策略**（tactics）将这个目标分解为更简单的子目标或直接证明该目标。当所有的子目标都被完全证明时，该分解过程就结束了。

定义 8 **目标：** 一个**目标**包含两条信息：基于局部上下文 Γ 和 Γ 的合式类型 P。

如果 E 是当前环境，那么我们结合上下文 Γ 和 P 可以将目标写作“$E, \Gamma \vdash^? P$”。当证明这样一个目标时，我们就是要构造 P 的一个证明，即基于环境 E 和上下文 Γ 的合式项 t。这样的一个项 t 被称为这个目标的一个**解**。

定义 9 **证明策略**（tactics）：证明策略是可以应用于目标的命令。它们能生成一个新的目标链（可能为空）。如果 g 是输入目标，g_1，...，g_k 是输出目标，那么该证明策略有一个相关联的函数，该函数可以从目标 g_i 的解中构造出 g 的解。

证明策略的概念可以追溯到非常古老的证明辅助工具；在 *Coq* 之前已有 *LCF* [49]、*HOL* [48] 和 *Isabelle* [73]。

3.1.3 第一个目标制导的证明

我们仍以公式 `(P→Q)→(Q→R)→(P→R)`为例，使目标和证明策略的概念更直观些。

3.1.3.1 声明命题变量

这一章中的例子最多使用四个命题变量。我们先在一个 section 中用类型 `Prop` 来声明这些变量，该 section 在章末结束。

```
Section Minimal_propositional_logic.
 Variables P Q R T : Prop.
```

P is assumed
Q is assumed
R is assumed
T is assumed

3.1.3.2 激活目标制导的证明

Theorem 命令用于陈述想要证明的定理，它指明该定理的名字和命题。在该命令之后可以使用一个可选命令 Proof，以使该会话的脚本更易阅读。然后结合当前上下文和命题，系统会创建一个原始目标。在显示目标时，用一条水平线，将上下文和命题上下分隔开。在下面的例子中，定理的名字为imp_trans。

```
Theorem imp_trans : (P→Q)→(Q→R)→P→R.
```
1 subgoal

```
  P : Prop
  Q : Prop
  R : Prop
  T : Prop
  ============================
   (P→Q)→(Q→R)→P→R
```
```
Proof.
```

Lemma 命令和 Theorem 相似，只是它一般用于中间的辅助结果。

3.1.3.3 引入新的假设

使用 intros 证明策略，我们将构造原始命题的证明转化为在增加了假设：H:P→Q，H':Q→R 和 p:P 的上下文中证明 R。这样该策略与蕴涵的 Heyting 解释相关：给出上下文中的蕴含假设，我们只需解释如何运用这些假设去构造结论的证明。2.2.3.2 节中给出的类型规则 *Lam* 证明了这是正确的。证明策略 intros 的参数是我们给不同假设起的名字。

```
intros H H' p.
```
1 subgoal

```
  P : Prop
  Q : Prop
  R : Prop
  T : Prop
  H : P→Q
  H' : Q→R
  p : P
  ============================
   R
```

注意这个策略既简化了要证的命题，同时又因为它在上下文中增加了新的假设，所以也增加了用于构造证明的可用资源。

3.1.3.4 应用假设

在当前的目标中，我们要证明的命题和假设 H' 的结论相同。通过证明策略 apply，我们可以利用该假设推进证明过程。该策略将一个项作为参数，它的类型可以分解为前提（箭头的左边）和结论（箭头的右边）。该项的结论要与待证明的命题相匹配，并将其前提作为一个新的目标。这里，我们使用命令“apply H'”产生新目标 Q，即 H' 的唯一前提。

```
apply H'.
```
1 subgoal

$P : Prop$
$Q : Prop$
$R : Prop$
$T : Prop$
$H : P \to Q$
$H' : Q \to R$
$p : P$
==========
Q

可以用同样的方法来求解这个新目标，即应用假设 H 来得到 Q 的证明，同时生成一个新目标 P。

```
apply H.
```
1 subgoal

$P : Prop$
$Q : Prop$
$R : Prop$
$T : Prop$
$H : P \to Q$
$H' : Q \to R$
$p : P$
==========
P

当假设中有多个箭头时，就会有多个前提，并且对每个前提生成一个新的目标。

3.1.3.5 直接使用假设

现在我们观察到要证明的命题正好是假设 p。应用策略 assumption 可以成功地识别这一事实，并且不再生成任何新目标。应用该策略后，这里就没有待证明的目标了，从而完成了证明。*Coq* 系统给出了相应的信息：

```
 assumption.
Proof completed.
```

3.1.3.6 建立并保存证明项

每个证明都要用 Qed 命令来结束。该命令会建立对应于策略序列的证明项，检测该项的类型是初始待证命题，并将该定理保存为一个定义。该定义将定理名，命题（定理的类型）和证明项关联起来。为了用户方便，*Coq* 系统会显示求得该解的策略序列。为了日后重用，该序列也可以被保存到文件中。使用 Print 命令可以将证明像 *Gallina* 定义一样显示出来。

```
Qed.
intros H H' p.
apply H'.
apply H.
assumption.
imp_trans is defined

Print imp_trans.
imp_trans = fun (H:P→Q)(H':Q→R)(p:P) ⇒ H' (H p)
    : (P→Q)→(Q→R)→P→R
```

3.1.3.7 阅读证明项

Coq 的证明项很难阅读，但是可以将其翻译为自然语言：每个抽象可以读作假定一个事实，每个项到变量的作用可以读作应用一个定理。图 3.2 给出了这样的一个翻译文本。

3.1.3.8 一步到位

前面介绍的证明有意地给出了证明 imp_trans 的全部细节。这是为了基于一个简单的例子，介绍基本策略的使用方法。然而用户不必每次都像这样给出证明的细节。事实上，一些自动证明策略能够一步到位证明这类目标。例如，imp_trans 可以这样证明：

```
Theorem imp_trans : (P→Q)→(Q→R)→P→R.
```

(H) : 假设 P→Q
(H') : 假设 Q→R
(p) : 假设 P
(1) : 通过使用 *(p)* 我们得到 P
(2) : 通过应用 *(H)* 我们得到 Q
(3) : 通过应用 *(H')* 我们得到 R
• : 消除 *H, H'* 和 *p*，我们可以证明 (P→Q)→(Q→R)→P→R

图 3.2 用英语解释证明

```
Proof.
  auto.
Qed.
```

3.2 类型规则和证明策略的联系

这一节将详细介绍如何用带简单类型的 λ-演算来描述命题逻辑，同时也将揭示基本证明策略如何与类型规则的简单应用相对应。

3.2.1 命题构造规则

类型规则不仅可以用来构造简单规范说明（参见 2.5.1 节），而且可以用于控制命题的构造。它们的区别仅在于用 Prop 代替了 Set。

3.2.1.1 命题变量

一个命题变量，即一个类型为 Prop 的变量。当标识符 id 被声明成 Prop 类型时，根据 **Var**（参见 2.2.3 节）规则，可以保证下面的类型判定成立：

$$E, \Gamma \vdash id : \texttt{Prop}.$$

在本章中，我们可以用 Check 命令来验证上面的判定：

```
Check Q.
```
Q : Prop

3.2.1.2 构造蕴含

前面提到过，蕴含在 *Coq* 系统中用箭头 → 表示；换句话说，如果有命题 P 和 Q，那么“P 蕴含 Q”可以由类型 $P \to Q$ 表示。这种构造类型的方法由下面的类型规则表示：

$$\textbf{Prod-Prop}\ \frac{E, \Gamma \vdash P : \texttt{Prop} \quad E, \Gamma \vdash Q : \texttt{Prop}}{E, \Gamma \vdash P \to Q : \texttt{Prop}}.$$

练习 3.1 分析 *Coq* 为如下命令所做的类型计算，重组它所对应的类型判断序列：

```
Check ((P→Q)→(Q→R)→P→R).
```
(P→Q)→(Q→R)→P→R : Prop

3.2.1.3 箭头规则

规则 **Prod-Set** 和 **Prod-Prop** 的不同仅在于，一个使用大类 `Set`，一个使用大类 `Prop`。这两条规则可以由参数为大类 s 的一条规则表示：

$$\textbf{Prod}\ \frac{E,\Gamma \vdash A : s \quad E,\Gamma \vdash B : s \quad s \in \{\texttt{Set},\texttt{Prop}\}}{E,\Gamma \vdash A \to B : s}$$

3.2.2 推理规则和证明策略

已知 P 是一个命题，且上下文或环境中包含形如 $x : P$ 的声明，那么 x 是 P 的一个证明项。这是 **Var** 规则的直接推论，此规则在 2.2.3 节中用于函数式程序设计：

$$\textbf{Var}\ \frac{(x : P) \in E \cup \Gamma}{E,\Gamma \vdash x : P}$$

因为如此，应用证明策略“`exact` x”可以立即得到目标的解，而不需要生成新的子目标。如果“$x : P$”是一个假设，即当前上下文中的一个局部声明，那么用无参数的证明策略 `assumption` 也可以得到目标的解。实际上，这条证明策略先遍历上下文来寻找相关假设，如果成功，则直接应用规则 **Var**。这条证明策略的优势在于，它不需要知道假设的名字。而当上下文中不包含满足条件的假设时，`assumption` 证明策略将会失败。

下面的例子显示了 `assumption` 策略的用法：

```
Section example_of_assumption.
 Hypothesis H : P→Q→R.

 Lemma L1 : P→Q→R.
 Proof.
  assumption.
 Qed.
End example_of_assumption.
```

如果不能使用 `assumption`，比如声明 $x : P$ 只能在全局环境中被找到，则必须使用策略“`exact` x”。

若遗失了定理或公理名，则可以使用 *Coq* 系统提供的查找工具找到它们（参见 5.1.3.4 节）。自动证明策略同样可以进行查找，但是它们使用自己的数据库，需要用户进行配置（参见 7.2.1 节）。

最后进行总结，我们知道，exact 策略不仅只是标识符，而且可以把具有正确类型的任何项作为参数。举一个极端的例子，如果我们可以直接构造整个证明项，那么直接将其作为参数，使用 exact 策略，一步得到完整的证明，下面是一个示例：

```
Theorem delta : (P→P→Q)→P→Q.
 Proof.
 exact (fun (H:P→P→Q)(p:P) ⇒ H p p).
Qed.
```

Coq 系统为此提供了 Proof 命令的一种变体。当使用该变体时，不必再用 Qed 来终止证明。前面的证明可以由下面的证明代替：

```
Theorem delta : (P→P→Q)→P→Q.
 Proof (fun (H:P→P→Q)(p:P) ⇒ H p p).
```

3.2.2.1 Modus Ponens 规则

大多数逻辑中都使用 *modus ponens* 推理规则（又称为**蕴涵消去**（implication elimination）**规则**），它根据 $P{\to}Q$ 和 P 的证明，构造一个 Q 的证明。在 Coq 系统中，这条规则可以直接由 **App** 规则（参见 2.2.3 节）实现。有了这条规则，Q 的证明，事实上可以看成把 $P{\to}Q$ 的证明 t 作用到 P 的证明 t' 上的结果：

$$\mathbf{App}\ \frac{E,\Gamma \vdash t : P{\to}Q \quad E,\Gamma \vdash t' : P}{E,\Gamma \vdash t\ t' : Q}$$

在交互式证明中，如果要证明 Q，且存在 $P{\to}Q$ 类型的一个项 t，那么可以使用证明策略“apply t”，生成目标 P。如果找到目标 P 的一个解 t'，那么这一步骤构造出来的证明为“$t\ t'$”。对于 3.1.3.4 节介绍的示例，使用了两次 apply 证明策略。

值得注意的是，自然语言中也将证明和程序一致对待：我们很自然地说“应用一个定理”和“应用一个函数”。这两种情况，事实上都是用的 **App** 规则。

我们经常会用到一些项，其类型由多个嵌套的箭头构成，比如，带有多个参数的函数或有多个前提的证明。在这种情况下，apply 策略把项作为一个整体，并且为每个前提生成一个目标。

apply 策略这一行为促使我们可以去定义一些关于项的类型的新概念：

定义 10 **首类型，终极类型**：如果项 t 具有类型 $P_1{\to}\ldots{\to}P_n{\to}Q$，那么项 $P_k{\to}\ldots{\to}P_n{\to}Q$ 和 Q 被称为 t **的秩为** k（*rank* k）**的首类型**。若 Q 不是箭头类型，则它也被称为**终极类型**。

我们在后面的章节中将会看到首类型和终极类型在其他证明策略，如 elim 策略中（参见 6.1.3 节）以及搜索命令（参见 5.1.3.4 节）中的作用。

已知 t 是一个项，其类型为：$P_1{\to}P_2{\to}\ldots{\to}P_n{\to}Q$，目标类型为

$$P_k{\to}P_{k+1}{\to}\ldots{\to}P_n{\to}Q$$