课后阅读：Coq 标准库中的列表

1 Coq 标准库中的 list

在 Coq 中，`list X` 表示一列 `X` 类型的元素，在标准库中，这一类型是通过 Coq 归纳类型定义的。下面介绍它的定义方式，重要函数与性质。此处为了演示这些函数的定义方式以及从定义出发构建各个性质的具体步骤重新按照标准库定义了 `list` ，下面的所有定义和性质都可以从标准库中找到。

```
Inductive list (X: Type): Type :=
  | nil: list X
  | cons (x: X) (l: list X): list X.
```

这里，`nil` 表示，这是一个空列；`cons` 表示一个非空列由一个元素（头元素）和另外一列元素（其余元素）构成，因此 `list` 可以看做用归纳类型表示的树结构的退化情况。下面是两个整数列表 `list Z` 的例子。

```
Check (cons Z 3 (nil Z)).
Check (cons Z 2 (cons Z 1 (nil Z))).
```

Coq 中也可以定义整数列表的列表，`list (list Z)` 。

```
Check (cons (list Z) (cons Z 2 (cons Z 1 (nil Z))) (nil (list Z))).
```

我们可以利用 Coq 的隐参数机制与 `Arguments` 指令，让我们省略 `list` 定义中的类型参数。

```
Arguments nil {X}.
Arguments cons {X} _ _.
```

例如，我们可以重写上面这些例子:

```
Check (cons 3 nil).
Check (cons 2 (cons 1 nil)).
Check (cons (cons 2 (cons 1 nil)) nil).
```

Coq 标准库还提供了一些 `Notation` 让 `list` 相关的描述变得更简单。目前，读者不需要了解或掌握相关声明的语法规定。

```
Notation "x :: y" := (cons x y)
  (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y []) ..).
```

下面用同一个整数列表的三种表示方法来演示这些 `Notation` 的使用方法。

```
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
Definition mylist2 := 1 :: 2 :: 3 :: nil.
Definition mylist3 := [1; 2; 3].
```

总的来说，我们在 Coq 中利用归纳类型 `list` 定义了我们日常理解的“列表”。这种定义方式从理论角度看是完备的，换言之，每一个 `A` 类型对象构成的（有穷长）列表都有一个 `list A` 类型的元素与之对应，反之亦然。但是，许多我们直观看来“列表”显然应当具备的性质，在 Coq 中都不是显然成立的，它们需要我们在 Coq 中利用归纳类型相关的方式做出证明；同时，所有“列表”相关的变换、函数与谓词，无论其表达的意思多么简单，它们都需要我们在 Coq 中利用归纳类型相关的方式做出定义。

下面介绍一些关于 `list` 的常用函数。根据 Coq 的要求，归纳类型上的递归函数必须依据归纳定义的方式进行递归。换言之，要定义 `list` 类型上的递归函数，就要能够计算这个 `list` 取值为 `nil` 的结果，并将这个 `list` 取值为 `cons a l` 时的结果规约为取值为 `l` 时的结果。很多关于列表的直观概念，都需要通过这样的方式导出，例如列表的长度、列表第 i 项的值、两个列表的连接等等。

下面定义的函数 `app` 表示列表的连接。

```
Fixpoint app {A: Type} (l1 l2: list A): list A :=
  match l1 with
  | nil        => l2
  | cons a l1' => cons a (app l1' l2)
  end.
```

在 Coq 中一般可以用 `++` 表示 `app` ，这个符号的结合性被规定为右结合。

```
Notation "x ++ y" := (app x y)
  (right associativity, at level 60).
```

以我们的日常理解看，“列表连接”的含义是不言自明的，就是把两个列表“连起来”。例如：

```
  [1; 2; 3] ++ [4; 5] = [1; 2; 3; 4; 5]
  [0; 0] ++ [0; 0; 0] = [0; 0; 0; 0; 0]
```

在 Coq 标准库定义 `app` 时，`[1; 2; 3]` 与后续列表的连接被规约为 `[2; 3]` 与后续列表的连接，又进一步规约为 `[3]` 与后续列表的连接，以此类推。

```
  [1; 2; 3] ++ [4; 5] =
  1 :: ([2; 3] ++ [4; 5]) =
  1 :: (2 :: ([3] ++ [4; 5])) =
  1 :: (2 :: (3 :: ([] ++ [4; 5]))) =
  1 :: (2 :: (3 :: [4; 5])) =
  [1; 2; 3; 4; 5]
```

上面式子中打上阴影的部分，就是每一次的递归调用。

我们可以在 Coq 中写下一些例子，检验上面定义的 `app` （也是 Coq 标准库中的 `app` ）确实定义了列表的连接。

```
Example test_app1: [1; 2; 3] ++ [4; 5] = [1; 2; 3; 4; 5].
Proof. reflexivity.  Qed.
```

```
Example test_app2: [2] ++ [3] ++ [] ++ [4; 5] = [2; 3; 4; 5].
Proof. reflexivity.  Qed.
```

```
Example test_app3: [1; 2; 3] ++ nil = [1; 2; 3].
Proof. reflexivity.  Qed.
```

上面定义 `app` 时我们只能使用 Coq 递归函数，下面要证明关于它的性质，我们也只能从 Coq 中的依据定义证明、结构归纳法证明与分类讨论证明开始。等证明了一些关于它的基础性质后，才可以使用这些性质证明进一步的结论。

我们熟知，空列表与任何列表连接，无论空列表在左还是空列表在右，都会得到这个列表本身。这可以在 Coq 中写作下面两个定理。

```
Theorem app_nil_l: forall A (l: list A), [] ++ l = l.
Proof. intros. simpl. reflexivity. Qed.
```

```
Theorem app_nil_r: forall A (l: list A), l ++ [] = l.
Proof.
  intros.
  induction l as [| n l' IHl'].
  + reflexivity.
  + simpl.
    rewrite -> IHl'.
    reflexivity.
Qed.
```

其中，`app_nil_l` 的证明只需使用 `app` 的定义化简 `[] ++ l` 即可。而 `app_nil_r` 的证明需要对列表作结构归纳。不过，这一归纳证明的思路是很简单的，以

```
[1; 2; 3] ++ []
```

的情况为例，归纳步骤的证明如下：

```
[1; 2; 3] ++ [] =
(1 :: [2; 3]) ++ [] =
1 :: ([2; 3] ++ []) =
1 :: ([2; 3]) =
[1; 2; 3]
```

其中阴影部分所指出的变换就是归纳假设。

我们还熟知，列表的连接具有结合律，在 Coq 中这一性质可以写作如下定理。

```
Theorem app_assoc:
  forall A (l1 l2 l3: list A),
    l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3.
```

若要证明这一定理，无非是选择对 `l1` 做归纳、对 `l2` 做归纳还是对 `l3` 做归纳证明。无论选择哪一种，结构归纳证明中的奠基步骤都没有问题，只需使用前面证明的 `app_nil_l` 与 `app_nil_r` 即可：

```
[] ++ (l2 ++ l3) = l2 ++ l3 = ([] ++ l2) ++ l3
l1 ++ ([] ++ l3) = l1 ++ l3 = (l1 ++ []) ++ l3
l1 ++ (l2 ++ []) = l1 ++ l2 = (l1 ++ l2) ++ []
```

然而，三种归纳证明的选择中，只有对 `l1` 归纳我们才能顺利地完成归纳步骤的证明。因为 `app` 的定义是对左侧列表做结构递归定义的，所以我们不难写出下面变换：

```
(a :: l1) ++ (l2 ++ l3) =
a :: (l1 ++ (l2 ++ l3))
```

```
((a :: l1) ++ l2) ++ l3) =
(a :: (l1 ++ l2)) ++ l3) =
a :: ((l1 ++ l2) ++ l3))
```

这样一来，我们就可以利用归纳假设完成归纳步骤的证明了。相反，如果采取对 `l2` 归纳证明或对 `l3` 归纳证明的策略，那么我们对于形如：

```
(l1 ++ (a :: l2)) ++ l3
(l1 ++ l2) ++ (a :: l3)
```

就束手无策了！尽管我们知道 `l1 ++ (a :: l2) = (l1 ++ [a]) ++ l2` ，但是我们没有在 Coq 中证明过这一性质，因而也就无法在此使用这样的性质做证明。

我们把上面总结的“对 `l1` 做结构归纳证明”的策略写成 Coq 证明如下。

```
Proof.
  intros.
  induction l1; simpl.
  + reflexivity.
  + rewrite -> IHl1.
    reflexivity.
Qed.
```

下面这条性质与前面所证明的性质有所不同，它从两个列表连接的结果反推两个列表。这条 Coq 定理说的是：如果 `l1 ++ l2` 是空列表，那么 `l1` 与 `l2` 都是空列表。

```
Theorem app_eq_nil:
  forall A (l1 l2: list A),
    l1 ++ l2 = [] ->
    l1 = [] /\ l2 = [].
```

要证明这一条性质，比较常见的思路是先利用反证法证明 `l1 = []` ，再利用这一结论证明 `l2 = []` 。具体而言，在利用反证法证明 `l1 = []` 时，我们先假设

```
l1 = a :: l1'
```

由此不难推出：

```
l1 ++ l2 = (a :: l1') ++ l2 = a :: (l1' ++ l2)
```

这与 `l1 ++ l2 = []` 是矛盾的。因此， `l1 = []` 。进一步，由 `app` 的定义又可知，

```
l1 ++ l2 = [] ++ l2 = l2
```

根据 `l1 ++ l2 = []` 的条件，这就意味着 `l2 = []` 。这样我们就证明了全部的结论。下面是上述证明思路在 Coq 中的表述。Coq 证明中我们并没有真正采用反证法，事实上，我们使用了 Coq 中基于归纳类型的分类讨论证明。具体而言，我们在这段证明中对 `l1` 的结构做了分类讨论：当 `l1` 为空列表时，我们证明 `l2` 也必须是空列表；当 `l1` 为非空列表时，我们推出矛盾。

```
Proof.
  intros.
  destruct l1.
  + (** 当_[l1]_为空列表时，需由_[[] ++ l2 = []]_证明_[[] = [] /\ l2 = []]_。*)
    simpl in H.
    tauto.
  + (** 当_[l1]_非空时，需要推出矛盾。*)
    simpl in H.
    discriminate H.
Qed.
```

到此为止，我们介绍了列表连接函数 `app` 的定义与其重要基础性质的证明。类似的，可以定义 Coq 递归函数 `rev` 表示对列表取反，并证明它的基础性质。

```
Fixpoint rev {A: Type} (l: list A) : list A :=
  match l with
  | nil       => nil
  | cons a l' => rev l' ++ [a]
  end.
```

```
Example test_rev1: rev [1; 2; 3] = [3; 2; 1].
Proof. reflexivity. Qed.
```

```
Example test_rev2: rev [1; 1; 1; 1] = [1; 1; 1; 1].
Proof. reflexivity. Qed.
```

```
Example test_rev3: @rev Z [] = [].
Proof. reflexivity. Qed.
```

下面几个性质的证明留作习题。

习题 1.

```
Theorem rev_app_distr:
  forall A (l1 l2: list A),
    rev (l1 ++ l2) = rev l2 ++ rev l1.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 2.

```
Theorem rev_involutive:
  forall A (l: list A), rev (rev l) = l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

如果熟悉函数式编程，不难发现，上面的 `rev` 定义尽管在数学是简洁明确的，但是其计算效率是比较低的。相对而言，利用下面的 `rev_append` 函数进行计算则效率较高。

```
Fixpoint rev_append {A} (l1 l2: list A): list A :=
  match l1 with
  | []       => l2
  | a :: l1' => rev_append l1' (a :: l2)
  end.
```

下面以 `[1; 2; 3; 4]` 的取反为例做对比。

```
  rev [1; 2; 3; 4] =
  rev [2; 3; 4] ++ [1] =
  (rev [3; 4] ++ [2]) ++ [1] =
  ((rev [4] ++ [3]) ++ [2]) ++ [1] =
  (((rev [] ++ [4]) ++ [3]) ++ [2]) ++ [1] =
  ((([] ++ [4]) ++ [3]) ++ [2]) ++ [1] =
  [4; 3; 2; 1]
```

```
  rev_append [1; 2; 3; 4] [] =
  rev_append [2; 3; 4] [1] =
  rev_append [3; 4] [2; 1] =
  rev_append [4] [3; 2; 1] =
  rev_append [] [4; 3; 2; 1] =
  [4; 3; 2; 1]
```

看上去两个函数的计算过程都包含四个递归步骤，但是 `rev` 的计算中还需要计算列表的连接“`++`”，因此它的总时间复杂度更高。下面证明 `rev` 与 `rev_append` 的计算结果相同，而证明的关键就是使用加强归纳法先证明下面引理。

```
Lemma rev_append_rev:
  forall A (l1 l2: list A),
    rev_append l1 l2 = rev l1 ++ l2.
Proof.
  intros.
  revert l2; induction l1 as [| a l1 IHl1]; intros; simpl.
  + reflexivity.
  + rewrite IHl1.
    rewrite <- app_assoc.
    reflexivity.
Qed.
```

利用这一引理，就可以直接证明下面结论了。

```
Theorem rev_alt:
  forall A (l: list A), rev l = rev_append l [].
Proof.
  intros.
  rewrite rev_append_rev.
  rewrite app_nil_r.
  reflexivity.
Qed.
```

下面再介绍一个关于列表的常用函数 `map`，它表示对一个列表中的所有元素统一做映射。它是一个 Coq 中的高阶函数。

```
Fixpoint map {X Y: Type} (f: X -> Y) (l: list X): list Y :=
  match l with
  | nil       => nil
  | cons x l' => cons (f x) (map f l')
  end.
```

这是一些例子：

```
Example test_map1: map (fun x => x - 2) [7; 5; 7] = [5; 3; 5].
Proof. reflexivity. Qed.
```

```
Example test_map2: map (fun x => x * x) [2; 1; 5] = [4; 1; 25].
Proof. reflexivity. Qed.
```

```
Example test_map3: map (fun x => [x]) [0; 1; 2; 3] = [[0]; [1]; [2]; [3]].
Proof. reflexivity. Qed.
```

很自然，如果对两个函数的复合做 `map` 操作，就相当于先后对这两个函数做 `map` 操作。这在 Coq 中可以很容易地利用归纳法完成证明。

```
Theorem map_map:
  forall X Y Z (f: Y -> Z) (g: X -> Y) (l: list X),
    map f (map g l) = map (fun x => f (g x)) l.
(* 证明详见Coq源代码。 *)
```

关于 `map` 的其他重要性质的证明，我们留作习题。

习题 3. 请证明下面关于 `map` 的性质。

```
Theorem map_app:
  forall X Y (f: X -> Y) (l l': list X),
    map f (l ++ l') = map f l ++ map f l'.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 4. 请证明下面关于 `map` 的性质。

```
Theorem map_rev:
  forall X Y (f: X -> Y) (l: list X),
    map f (rev l) = rev (map f l).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 5. 请证明下面关于 `map` 的性质。

```
Theorem map_ext:
  forall X Y (f g: X -> Y),
    (forall a, f a = g a) ->
    (forall l, map f l = map g l).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 6. 请证明下面关于 `map` 的性质。

```
Theorem map_id:
  forall X (l: list X), map (fun x => x) l = l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

除了提供一系列关于列表的常用函数之外，Coq 标准库还提供了不少关于列表的谓词。这里我们只介绍其中最常用的一个：`In` 。

```
Fixpoint In {A: Type} (a: A) (l: list A): Prop :=
  match l with
  | nil => False
  | b :: l' => b = a \/ In a l'
  end.
```

根据这一定义，`In a l` 表示 `a` 是 `l` 的一个元素。可以证明 `In a l` 的充分必要条件是 `l` 可以写成 `l1 ++ a :: l2` 的形式。

首先是充分性。要由 `l = l1 ++ a :: l2` 推出 `In a l` 可以直接写作下面更简单的形式。

```
Theorem in_elt:
  forall A (a: A) (l1 l2: list A),
    In a (l1 ++ a :: l2).
```

证明时，对 `l1` 使用归纳证明上面命题，也比对 `l` 归纳证明 `l = l1 ++ a :: l2` 能推出 `In a l` 来得方便。下面是 Coq 证明。

```
Proof.
  intros.
  induction l1 as [| b l1 IHl1]; simpl.
  + tauto.
  + tauto.
Qed.
```

在证明必要性之前，我们先证明一个引理：如果 `a` 出现在 `l` 中，那么 `a` 也出现在 `b::l` 中。

```
Lemma elt_cons:
  forall A (a b: A) (l: list A),
    (exists l1 l2, l = l1 ++ a :: l2) ->
    (exists l1 l2, b :: l = l1 ++ a :: l2).
Proof.
  intros A a b l [l1 [l2 H]].
  exists (b :: l1), l2.
  rewrite H.
  reflexivity.
Qed.
```

在上面证明中，我们利用前提 `H: l = l1 ++ a :: l2` 以及 `rewrite H` 指令将结论中的 `l` 变为了 `l1 ++ a :: l2`。在 Coq 证明中，我们经常需要利用等式将一个变量替换为与它相等的式子，有时可能还需要在前提和结论中替换多处。Coq 提供了 `subst` 指令用于完成这样的变换。

```
Lemma elt_cons_again:
  forall A (a b: A) (l: list A),
    (exists l1 l2, l = l1 ++ a :: l2) ->
    (exists l1 l2, b :: l = l1 ++ a :: l2).
Proof.
  intros A a b l [l1 [l2 H]].
  exists (b :: l1), l2.
  subst l.
  (** 在执行完_[subst l]_指令后，结论中的_[l]_被替换为了_[l1 ++ a :: l2]_并且前提中
      的_[H]_以及_[l]_都被删除了。*)
  reflexivity.
Qed.
```

下面证明必要性定理。

```
Theorem in_split:
  forall A (a: A) (l: list A),
    In a l ->
    exists l1 l2, l = l1 ++ a :: l2.
Proof.
  intros.
  induction l as [| b l IHl]; simpl in *.
  + (** 奠基步骤是_[l]_为空列表的情况，此时可以直接由_[In a l]]_推出矛盾。*)
    tauto.
  + (** 归纳步骤是要由_[In a l]_情况下的归纳假设推出_[In a (b :: l)]_时的结论。下面
        先对_[In a (b :: l)]_这一前提成立的原因做分类讨论。*)
    destruct H.
    - (** 情况一：_[b = a]_。这一情况下结论是容易证明的，我们将使用_[subst b]_指令将
          _[b]_都替换为_[a]_。 *)
      exists nil, l.
      subst b; reflexivity.
    - (** 情况二：_[In a l]_。这一情况下可以使用归纳假设与_[elt_cons]_引理完成证明。*)
      specialize (IHl H).
      apply elt_cons, IHl.
Qed.
```

Coq 证明脚本 1. subst 指令. 如果 `x` 是一个 Coq 变量，证明指令 `subst x` 表示先找一个具有形式 `x = ...` 或 `... = x` 的前提，再将证明前提和结论中的所有 `x` 全部根据这一等式替换称为与 `x` 相等的表达式。该指令执行结束后，`x` 变量与这个等式都会被从前提中删除。注意，这一等式的另一侧中不得包含 `x` 本身。另外，`subst` 指令也可以对多个变量进行替换操作。例如 `subst x y` 就表示先执行 `subst x` 再执行 `subst y` 。最后，`subst` 指令也可以不带参数，这就表示对所有证明环境中的变量都尝试利用前提中的等式进行替换。

下面再列举几条关于 `In` 的重要性质。它们的 Coq 证明都可以在 Coq 代码中找到。首先，`l1 ++ l2` 的元素要么是 `l1` 的元素要么是 `l2` 的元素；`rev l` 的元素全都是 `l` 的元素。

```
Theorem in_app_iff:
  forall A (l1 l2: list A) (a: A),
    In a (l1 ++ l2) <-> In a l1 \/ In a l2.
```

```
Theorem in_rev:
  forall A (l: list A) (a: A),
    In a l <-> In a (rev l).
```

接下去两条定理探讨了 `map f l` 中元素具备的性质，其中 `in_map` 给出了形式较为简洁的必要条件，而 `in_map_iff` 给出的充要条件其形式要复杂一些。

```
Theorem in_map:
  forall A B (f: A -> B) (l: list A) (a: A),
    In a l -> In (f a) (map f l).
```

```
Theorem in_map_iff:
  forall A B (f: A -> B) (l: list A) (b: B),
    In b (map f l) <->
    (exists a, f a = b /\ In a l).
```

以上介绍的列表相关定义域性质都可以在 Coq 标准库中找到。使用时，只需要导入 `Coq.Lists.List` 即可。

习题 7. 下面定义的 `suffixes` 函数计算了一个列表的所有后缀。

```
Fixpoint suffixes {A: Type} (l: list A): list (list A) :=
  match l with
  | nil => [nil]
  | a :: l' => l :: suffixes l'
  end.
```

例如

```
  suffixes []           = [ [] ]
  suffixes [1]          = [ [1]; [] ]
  suffixes [1; 2]       = [ [1; 2]; [2]; [] ]
  suffixes [1; 2; 3; 4] = [ [1; 2; 3; 4];
                            [2; 3; 4]   ;
                            [3; 4]      ;
                            [4]         ;
                            []          ]
```

接下去，请分三步证明，`suffixes l` 中的确实是 `l` 的全部后缀。

```
Lemma self_in_suffixes:
  forall A (l: list A), In l (suffixes l).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Theorem in_suffixes:
  forall A (l1 l2: list A),
    In l2 (suffixes (l1 ++ l2)).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Theorem in_suffixes_inv:
  forall A (l2 l: list A),
    In l2 (suffixes l) ->
    exists l1, l1 ++ l2 = l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 8. 下面定义的 `prefixes` 函数计算了一个列表的所有前缀。

```
Fixpoint prefixes {A: Type} (l: list A): list (list A) :=
  match l with
  | nil => [nil]
  | a :: l0 => nil :: (map (cons a) (prefixes l0))
  end.
```

例如：

```
  prefixes [1; 2]    = [ []     ;
                         [1]    ;
                         [1; 2] ]
```

```
  prefixes [0; 1; 2] = [] ::
                       map (cons 0 (prefixes [1; 2]))
                     = [] ::
                       [ 0 :: []     ;
                         0 :: [1]    ;
                         0 :: [1; 2] ]
                     = [ []        ;
                         [0]       ;
                         [0; 1]    ;
                         [0; 1; 2] ]
```

接下去，请分三步证明，`prefixes l` 中的确实是 `l` 的全部前缀。

```
Lemma nil_in_prefixes:
  forall A (l: list A), In nil (prefixes l).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Theorem in_prefixes:
  forall A (l1 l2: list A),
    In l1 (prefixes (l1 ++ l2)).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Theorem in_prefixes_inv:
  forall A (l1 l: list A),
    In l1 (prefixes l) ->
    exists l2, l1 ++ l2 = l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 9. 下面的 `sublists` 定义了列表中的所有连续段。

```
Fixpoint sublists {A: Type} (l: list A): list (list A) :=
  match l with
  | nil => [nil]
  | a :: l0 => map (cons a) (prefixes l0) ++ sublists l0
  end.
```

请证明 `sublists l` 的元素确实是 `l` 中的所有连续段。提示：必要时可以添加并证明一些前置引理帮助完成证明。

```
Theorem in_sublists:
  forall A (l1 l2 l3: list A),
    In l2 (sublists (l1 ++ l2 ++ l3)).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Theorem in_sublists_inv:
  forall A (l2 l: list A),
    In l2 (sublists l) ->
    exists l1 l3, l1 ++ l2 ++ l3 = l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

2 列表相关的证明技巧

在基于 `list` 的计算中，有两类常见的计算，一类是从左向右计算，一类是从右向左计算。以对整数序列求和为例，下面的 `sum_L2R` 刻画了从左向右的计算方法，而 `sum_R2L` 刻画了从右向左的计算方法。

```
Fixpoint sum_L2R_rec (l: list Z) (s: Z): Z :=
  match l with
  | nil       => s
  | cons z l' => sum_L2R_rec l' (s + z)
  end.
```

```
Definition sum_L2R (l: list Z): Z := sum_L2R_rec l 0.
```

```
Fixpoint sum_R2L (l: list Z): Z :=
  match l with
  | nil       => 0
  | cons z l' => z + sum_R2L l'
  end.
```

以对 `[1; 3; 5; 7]` 求和为例。

```
  sum_L2R [1; 3; 5; 7] =
  sum_L2R_rec [1; 3; 5; 7] 0 =
  sum_L2R_rec [3; 5; 7] (0 + 1) =
  sum_L2R_rec [5; 7] ((0 + 1) + 3) =
  sum_L2R_rec [7] (((0 + 1) + 3) + 5) =
  sum_L2R_rec [] ((((0 + 1) + 3) + 5) + 7) =
  (((0 + 1) + 3) + 5) + 7
```

```
  sum_R2L [1; 3; 5; 7] =
  1 + sum_R2L [3; 5; 7] =
  1 + (3 + sum_R2L [5; 7]) =
  1 + (3 + (5 + sum_R2L [7])) =
  1 + (3 + (5 + (7 + sum_R2L []))) =
  1 + (3 + (5 + (7 + 0)))
```

许多列表上的运算都可以归结为从左向右计算和从右向左计算。Coq 标准库把这样的通用计算模式刻画为 `fold_left` 与 `fold_right` 。

```
Fixpoint fold_left {A B: Type} (f: A -> B -> A) (l: list B) (a0: A): A :=
  match l with
  | nil       => a0
  | cons b l' => fold_left f l' (f a0 b)
  end.
```

```
Fixpoint fold_right {A B: Type} (f: A -> B -> B) (b0: B) (l: list A): B :=
  match l with
  | nil       => b0
  | cons a l' => f a (fold_right f b0 l')
  end.
```

仔细观察，不难发现 `sum_L2R` 与 `sum_R2L` 可以分别用 `fold_left` 与 `fold_right` 表示出来。下面是它们的对应关系。

```
Fact sum_L2R_rec_is_fold_left:
  forall (l: list Z) (s: Z),
    sum_L2R_rec l s = fold_left (fun z1 z2 => z1 + z2) l s.
```

```
Fact sum_L2R_is_fold_left:
  forall l: list Z,
    sum_L2R l = fold_left (fun z1 z2 => z1 + z2) l 0.
```

```
Fact sum_R2L_is_fold_right:
  forall l: list Z,
    sum_R2L l = fold_right (fun z1 z2 => z1 + z2) 0 l.
```

当然，我们都知道，根据加法结合律 `sum_L2R` 与 `sum_R2L` 应当相等。不过，我们无法直接证明这一结论。直接使用归纳法证明很快就会陷入困境。

```
Theorem sum_L2R_sum_R2L:
  forall (l: list Z),
    sum_L2R l = sum_R2L l.
Proof.
  intros.
  induction l.
  + (** 由于_[sum_L2R [] = sum_L2R_rec [] 0 = 0]_并且_[sum_R2L [] = 0]_，所以奠
       基步骤的结论显然成立。*)
    reflexivity.
  + (** 根据定义
       - sum_R2L (a :: l) = a + sum_R2L l
       - sum_L2R (a :: l) = sum_L2R_rec (a :: l) 0 = sum_L2R_rec l a
       于是后者无法被归结为关于_[sum_L2R l]_或者_[sum_L2R_rec l 0]_的式子，自然也
       就无法使用归纳假设证明_[sum_L2R (a :: l)]_与_[sum_R2L (a :: l)]_相等。*)
Abort.
```

一些读者会想到先证明一个形如 `sum_L2R_rec l a = a + sum_L2R l` 的引理从而完成上面的归纳证明。这是可行的，其中关键的归纳步骤是要证明，如果下面归纳假设成立

`forall s, sum_L2R_rec l s = s + sum_L2R_rec l 0`

那么

`sum_L2R_rec (a :: l) s = s + sum_L2R_rec (a :: l) 0` 。

根据定义和归纳假设我们知道：

```
  sum_L2R_rec (a :: l) s =
  sum_L2R_rec l (s + a) =
  (s + a) + sum_L2R_rec l 0
```

```
  s + sum_L2R_rec (a :: l) 0 =
  s + sum_L2R_rec l a =
  s + (a + sum_L2R_rec l 0)
```

这样就能完成归纳步骤的证明了。上述证明的 Coq 版本如下。

```
Lemma sum_L2R_rec_sum_L2R:
  forall (s: Z) (l: list Z),
    sum_L2R_rec l s = s + sum_L2R l.
Proof.
  intros.
  unfold sum_L2R.
  revert s; induction l; simpl; intros.
  + lia.
  + rewrite (IHl a), (IHl (s + a)).
    lia.
Qed.
```

基于此证明原先的定理 `sum_L2R_sum_R2L` 是容易的。

```
Theorem sum_L2R_sum_R2L:
  forall (l: list Z), sum_L2R l = sum_R2L l.
Proof.
  intros.
  induction l.
  + reflexivity.
  + unfold sum_L2R; simpl.
    rewrite sum_L2R_rec_sum_L2R.
    lia.
Qed.
```

上面的证明思路是从结论出发，尝试是否可以通过对 `l` 归纳证明 `sum_L2R l = sum_R2L l` ，并在证明中根据需要补充证明相关的引理。同样是要证明这一结论，还有下面这一种不同的证明方案，它的主要思路是从 `sum_L2R` 和 `sum_R2L` 两者定义的结构出发构造证明。在这两者的定义中， `sum_R2L` 和 `sum_L2R_rec` 都是对列表递归定义的函数，因此可以优先证明此二者之间的联系。

```
Lemma sum_L2R_rec_sum_R2L:
  forall (s: Z) (l: list Z),
    sum_L2R_rec l s = s + sum_R2L l.
Proof.
  intros.
  revert s; induction l; intros; simpl.
  + lia.
  + rewrite IHl.
    lia.
Qed.
```

在此基础上就可以导出 `sum_L2R` 和 `sum_R2L` 等价。

```
Theorem sum_L2R_sum_R2L_____second_proof:
  forall (l: list Z), sum_L2R l = sum_R2L l.
Proof.
  intros.
  unfold sum_L2R.
  rewrite sum_L2R_rec_sum_R2L.
  lia.
Qed.
```

回顾 `sum_L2R` 与 `sum_R2L` 的定义，其实称它们分别是从左向右计算和从右向左有两方面的因素。第一是从结果看：

```
sum_L2R [1; 3; 5; 7] = (((0 + 1) + 3) + 5) + 7
sum_R2L [1; 3; 5; 7] = 1 + (3 + (5 + (7 + 0)))
```

第二是从计算过程看，

```
sum_L2R [1; 3; 5; 7] =
sum_L2R_rec [1; 3; 5; 7] 0 =
sum_L2R_rec [3; 5; 7] (0 + 1) =
sum_L2R_rec [5; 7] ((0 + 1) + 3) =
sum_L2R_rec [7] (((0 + 1) + 3) + 5) =
sum_L2R_rec [] ((((0 + 1) + 3) + 5) + 7) =
(((0 + 1) + 3) + 5) + 7
```

上面 `sum_L2R` 的计算过程中，就从左向右依次计算了 `0` 、`0 + 1` 、`(0 + 1) + 3` 等等这些中间结果，而 `sum_R2L` 的计算过程就是从右向左的。这一对比也可以从 `sum_L2R` 与 `sum_R2L` 的定义看出。在 `sum_L2R_rec` 的递归定义中，加法运算出现在递归调用的参数中，也就是说，需要先计算加法运算的结果，再递归调用。由于 Coq 中 `list` 的定义是从左向右的归纳定义类型，因此，递归调用前进行计算就意味着计算过程是从左向右的。而 `sum_R2L` 的定义恰恰相反，它是将递归调用的结果与其他数值相加得到返回值，也就是说，需要先递归调用再做加法运算，因此，它的计算过程是从右向左的。下面图中的阴影部分代码把加法运算发生的位置标记出来了。

```
Fixpoint sum_L2R_rec (l: list Z) (s: Z): Z :=
  match l with
  | nil       => s
  | cons z l' => sum_L2R_rec l' (s + z)
  end.
```

```
Fixpoint sum_R2L (l: list Z): Z :=
  match l with
  | nil       => 0
  | cons z l' => z + sum_R2L l'
  end.
```

必须指出，从计算结果看和从计算过程看，是两种不同的视角。例如，我们还可以定义下面 Coq 函数用于表示整数列表的求和。

```
Fixpoint sum_R2L_by_L2R_rec (l: list Z) (cont: Z -> Z): Z :=
  match l with
  | nil     => cont 0
  | z :: l0 => sum_R2L_by_L2R_rec l0 (fun z0 => cont (z + z0))
  end.
```

```
Definition sum_R2L_by_L2R (l: list Z): Z :=
  sum_R2L_by_L2R_rec l (fun z => z).
```

它从计算过程看是从左向右计算，但是它从结果看是从右向左计算，例如：

```
sum_R2L_by_L2R [1; 3; 5; 7] =
sum_R2L_by_L2R_rec [1; 3; 5; 7] (fun z => z) =
sum_R2L_by_L2R_rec [3; 5; 7] (fun z => 1 + z) =
sum_R2L_by_L2R_rec [5; 7] (fun z => 1 + (3 + z)) =
sum_R2L_by_L2R_rec [7] (fun z => 1 + (3 + (5 + z))) =
sum_R2L_by_L2R_rec [] (fun z => 1 + (3 + (5 + (7 + z)))) =
1 + (3 + (5 + (7 + 0)))
```

它的计算过程中依次计算得到了

```
  (fun z => z)
  (fun z => 1 + z)
  (fun z => 1 + (3 + z))
  (fun z => 1 + (3 + (5 + z)))
  (fun z => 1 + (3 + (5 + (7 + z))))
```

上面这些从左到右的中间结果，但是它的最终计算结果却是从右向左的。

我们也可以证明这个定义与先前定义的 `sum_L2R` 与 `sum_R2L` 之间的关系。我们先归纳证明 `sum_R2L_by_L2R_rec` 与 `sum_R2L` 之间的关系，以及 `sum_R2L_by_L2R_rec` 与 `sum_L2R_rec` 之间的关系，再由这两者推导出 `sum_R2L_by_L2R` 与 `sum_L2R` 、 `sum_R2L` 之间相等。具体的 Coq 证明这里略去了，这里只列出结论。

```
Lemma sum_R2L_results_aux:
  forall (cont: Z -> Z) (l: list Z),
    cont (sum_R2L l) = sum_R2L_by_L2R_rec l cont.
```

```
Lemma sum_L2R_approaches_aux:
  forall (cont: Z -> Z) (s: Z) (l: list Z),
    (forall z, cont z = s + z) ->
    sum_L2R_rec l s = sum_R2L_by_L2R_rec l cont.
```

```
Theorem sum_R2L_results:
  forall l, sum_R2L l = sum_R2L_by_L2R l.
```

```
Theorem sum_L2R_approaches:
  forall l, sum_L2R l = sum_R2L_by_L2R l.
```

一般而言，要证明两项“从左向右”计算之间的关系比较容易，要证明两种“从右向左”计算之间的关系也比较容易。但是要证明“从左向右”与“从右向左”之间的关系往往就要复杂一些。像上面分析的那样，从结论出发，采用加强归纳法证明，或者从定义出发证明辅助递归定义之间的关系都是常见的证明思路。

回顾前面介绍的 `rev` 函数与 `rev_append` 函数，我们不难发现，其实 `rev` 是“从右向左”计算而 `rev_append` 是“从左向右”计算，而当我们证明它们计算结果相等的时候（`rev_alt` 定理）也采用了类似的加强归纳法。以这样的观点来看，`map` 函数与 `rev` 一样，是一个“从右向左”计算的函数。我们可以定义它的“从左向右”版本。

```
Fixpoint map_L2R_rec
           {X Y: Type}
           (f: X -> Y)
           (l: list X)
           (l': list Y): list Y :=
  match l with
  | nil        => l'
  | cons x0 l0 => map_L2R_rec f l0 (l' ++ [f x0])
  end.
```

```
Definition map_L2R {X Y: Type} (f: X -> Y) (l: list X): list Y :=
  map_L2R_rec f l [].
```

习题 10. 请分两步证明 `map_L2R` 与 `map` 的计算结果是相等的。

```
Lemma map_L2R_rec_map: forall X Y (f: X -> Y) l l',
  map_L2R_rec f l l' = l' ++ map f l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Theorem map_alt: forall X Y (f: X -> Y) l,
  map_L2R f l = map f l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 11. 请试着证明下面结论。第一小题将 `fold_left` 转化为 `fold_right` 。

```
Theorem fold_left_fold_right:
  forall {A B: Type} (f: A -> B -> A) (l: list B) (a0: A),
    fold_left f l a0 =
    fold_right (fun (b: B) (g: A -> A) (a: A) => g (f a b)) (fun a => a) l a0.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

第二小题是将 `fold_right` 转化为 `fold_left` 。提示：尽管这一小题看上去与第一小题是对称的，但是它证明起来要复杂很多，可能需要引入一条辅助引理才能完成证明。

```
Theorem fold_right_fold_left:
  forall {A B: Type} (f: A -> B -> B) (b0: B) (l: list A),
    fold_right f b0 l =
    fold_left (fun (g: B -> B) (a: A) (b: B) => g (f a b)) l (fun b => b) b0.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 12. 下面定义的 `list_inc` 定义了“整数列表单调递增”这个性质。这个定义分为两步。

```
Fixpoint list_inc_rec (a: Z) (l: list Z): Prop :=
  match l with
  | nil => True
  | cons b l0 => a < b /\ list_inc_rec b l0
  end.
```

```
Definition list_inc (l: list Z): Prop :=
  match l with
  | nil => True
  | cons a l0 => list_inc_rec a l0
  end.
```

例如：

```
  list_inc [] = True
```

```
  list_inc [x1] = list_inc_rec x1 []
                = True
```

```
  list_inc [x1; x2] = list_inc_rec x1 [x2]
                    = x1 < x2 /\ list_inc_rec x2 []
                    = x1 < x2 /\ True
```

```
  list_inc [x1; x2; x3] = list_inc_rec x1 [x2; x3]
                        = x1 < x2 /\ list_inc_rec x2 [x3]
                        = x1 < x2 /\ x2 < x3 /\ list_inc_rec x3 []
                        = x1 < x2 /\ x2 < x3 /\ True
```

下面请分两步证明，如果 `l1 ++ a1 :: a2 :: l2` 是单调递增的，那么必定有 `a1 < a2` 。

```
Lemma list_inc_rec_always_increasing':
  forall a l1 a1 a2 l2,
    list_inc_rec a (l1 ++ a1 :: a2 :: l2) ->
    a1 < a2.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Lemma list_inc_always_increasing':
  forall l1 a1 a2 l2,
    list_inc (l1 ++ a1 :: a2 :: l2) ->
    a1 < a2.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

除了 `list_inc` 之外，我们也可以采取下面这种方式定义“单调递增”。

```
Definition always_increasing (l: list Z): Prop :=
  forall l1 a1 a2 l2,
    l1 ++ a1 :: a2 :: l2 = l ->
    a1 < a2.
```

既然两种定义都表达了“单调递增”的意思，那么我们理应能够证明它们等价。先前的两个引理意味着我们已经可以使用 `list_inc` 推出 `always_increasing` 。这是它的 Coq 证明。

```
Theorem list_inc_always_increasing:
  forall l, list_inc l -> always_increasing l.
Proof.
  unfold always_increasing.
  intros.
  subst l.
  pose proof list_inc_always_increasing' _ _ _ _ H.
  tauto.
Qed.
```

下面请你证明，`always_increasing` 也能推出 `list_inc` 。提示：如果需要，你可以写出并证明一些前置引理用于辅助证明。

```
Theorem always_increasing_list_inc:
  forall l,
    always_increasing l -> list_inc l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 13. 下面定义的 `list_sinc` 和 `strong_increasing` 都表示整数序列中的每一个元素都比它左侧所有元素的和还要大。值得一提的是，这里递归定义的 `list_sinc_rec` 既不是单纯的从左向右计算又不是单纯的从右向左计算，它的定义中既有递归调用之前的计算，又有递归调用之后的计算。

```
Fixpoint list_sinc_rec (a: Z) (l: list Z): Prop :=
  match l with
  | nil => True
  | cons b l0 => a < b /\ list_sinc_rec (a + b) l0
  end.
```

```
Definition list_sinc (l: list Z): Prop :=
  match l with
  | nil => True
  | cons a l0 => 0 < a /\ list_sinc_rec a l0
  end.
```

```
Definition strong_increasing (l: list Z): Prop :=
  forall l1 a l2,
    l1 ++ a :: l2 = l ->
    sum_L2R l1 < a.
```

请你证明 `list_sinc` 与 `strong_increasing` 等价。提示：如果需要，你可以写出并证明一些前置引理用于辅助证明，也可以定义一些辅助概念用于证明。

```
Theorem list_sinc_strong_increasing:
  forall l, list_sinc l -> strong_increasing l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Theorem strong_increasing_list_sinc:
  forall l,
    strong_increasing l -> list_sinc l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

3 Coq 标准库中的 string

除了一般性的列表 `list` 之外，Coq 标准库中还定义了字符串这一数学对象。例如下面这些都是字符串。

```
Check "abc".
Check "Hello world".
Check "".
```

字符串集合 `string` 的定义与 `list` 十分相似：

```
Inductive string :=
| EmptyString: string
| String: (c: ascii) (s: string).
```

其中，`EmptyString` 就相当于 `nil`，`String` 就相当于 `cons`，`ascii` 表示所有 ascii 码的集合。因此，我们可以很容易的定义 `string` 与 `list` 之间的相互转化。双向的转化函数都可以用 Coq 结构递归函数定义。

```
Fixpoint list_ascii_of_string (s: string): list ascii :=
  match s with
  | EmptyString => nil
  | String c0 s0 => cons c0 (list_ascii_of_string s0)
  end.
```

```
Fixpoint string_of_list_ascii (l: list ascii): string :=
  match l with
  | nil => EmptyString
  | cons c0 l0 => String c0 (string_of_list_ascii l0)
  end.
```

在此基础上，我们又可以利用 Coq 中的结构归纳法证明这两个转化函数互为反函数。

```
Lemma string_of_list_ascii_of_string:
  forall s: string, string_of_list_ascii (list_ascii_of_string s) = s.
Proof.
  intros.
  induction s; simpl.
  + reflexivity.
  + rewrite IHs.
    reflexivity.
Qed.
```

```
Lemma list_ascii_of_string_of_list_ascii:
  forall l: list ascii, list_ascii_of_string (string_of_list_ascii l) = l.
Proof.
  intros.
  induction l; simpl.
  + reflexivity.
  + rewrite IHl.
    reflexivity.
Qed.
```

数学上，我们称一个函数 $f : A \to B$ 是一个双射，当且仅当

$$\forall a_1, a_2 \in A.\ f(a_1) = f(a_2) \Rightarrow a_1 = a_2$$
$$\forall b \in B.\ \exists a \in A.\ f(a) = b。$$

而在 Coq 中，往往会直接显式地写出函数和它的反函数，这样也便于在后续证明中使用 `rewrite` 重写指令证明。例如，要在上面两结论基础上，证明 `list_ascii_of_string` 是双射是容易的。

```
Lemma list_ascii_of_string_inj: forall s1 s2: string,
  list_ascii_of_string s1 = list_ascii_of_string s2 ->
  s1 = s2.
Proof.
  intros.
  rewrite <- (string_of_list_ascii_of_string s1).
  rewrite <- (string_of_list_ascii_of_string s2).
  rewrite H.
  reflexivity.
Qed.
```

```
Lemma list_ascii_of_string_surj: forall l: list ascii,
  exists s: string, list_ascii_of_string s =  l.
Proof.
  intros.
  exists (string_of_list_ascii l).
  apply list_ascii_of_string_of_list_ascii.
Qed.
```

上面提到的这些转化函数与反函数性质都是 Coq 标准库中提供的定义与证明。