Coq 中的结构归纳法证明

1 结构归纳法

我们接下去将证明一些关于 `tree_height` ， `tree_size` 与 `tree_reverse` 的基本性质。我们在证明中将会使用的主要方法是归纳法。

相信大家都很熟悉自然数集上的数学归纳法。数学归纳法说的是：如果我们要证明某性质 `P` 对于任意自然数 `n` 都成立，那么我可以将证明分为如下两步：

- 奠基步骤：证明 `P 0` 成立；
- 归纳步骤：证明对于任意自然数 `n` ，如果 `P n` 成立，那么 `P (n + 1)` 也成立。

对二叉树的归纳证明与上面的数学归纳法稍有不同。具体而言，如果我们要证明某性质 `P` 对于一切二叉树 `t` 都成立，那么我们只需要证明以下两个结论：

- 奠基步骤：证明 `P Leaf` 成立；
- 归纳步骤：证明对于任意二叉树 `l` `r` 以及任意整数标签 `n` ，如果 `P l` 与 `P r` 都成立，那么 `P (Node l n r)` 也成立。

这样的证明方法就称为结构归纳法。在 Coq 中， `induction` 指令表示：使用结构归纳法。下面是几个证明的例子。

第一个例子是证明 `tree_size` 与 `tree_reverse` 之间的关系。

```
Lemma reverse_size: forall t,
  tree_size (tree_reverse t) = tree_size t.
Proof.
  intros.
  induction t.
  (** 上面这个指令说的是：对_[t]_结构归纳。Coq会自动将原命题规约为两个证明目标，
      即奠基步骤和归纳步骤。*)
  + simpl.
    (** 第一个分支是奠基步骤。这个_[simpl]_指令表示将结论中用到的递归函数根据定
        义化简。*)
    reflexivity.
  + simpl.
    (** 第二个分支是归纳步骤。我们看到证明目标中有两个前提_[IHt1]_以及_[IHt2]_。
        在英文中_[IH]_表示induction hypothesis的缩写，也就是归纳假设。在这个证明
        中_[IHt1]_与_[IHt2]_分别是左子树_[t1]_与右子树_[t2]_的归纳假设。 *)
    rewrite IHt1.
    rewrite IHt2.
    lia.
Qed.
```

第二个例子很类似，是证明 `tree_height` 与 `tree_reverse` 之间的关系。

```
Lemma reverse_height: forall t,
  tree_height (tree_reverse t) = tree_height t.
Proof.
  intros.
  induction t.
  + simpl.
    reflexivity.
  + simpl.
    rewrite IHt1.
    rewrite IHt2.
    lia.
    (** 注意：_[lia]_指令也是能够处理_[Z.max]_与_[Z.min]_的。*)
Qed.
```

在上面的证明中，归纳证明的奠基步骤和归纳步骤都需要先用 `simpl` 指令对待证明结论化简。在 Coq 中，可以用分号 `;` 连接两个证明指令，表示先执行第一个证明指令，再在它产生的每一个证明目标中执行第二个指令。因此，我们就可以用 `induction t; simple` 简化上面证明。

```
Lemma reverse_height_attempt2: forall t,
  tree_height (tree_reverse t) = tree_height t.
Proof.
  intros.
  induction t; simpl.
  (** 这条指令执行后，两个证明目标中都做了化简。*)
  + reflexivity.
  + lia.
Qed.
```

Coq 证明脚本 1. 分号 `;` . 在 Coq 证明语言中可以用分号将小的证明指令连接起来形成大的证明指令，某种意义上说，Coq 证明语言也是一种程序语言，其中的分号表示了普通程序语言中顺序执行的意思。只不过，每一个 Coq 证明指令可能产生一个规约后的证明目标，也可能产生超过一个规约后的证明目标。`tac1 ; tac2` 这个证明指令表示先执行指令 `tac1` ，再对于 `tac1` 生成的每一个证明目标执行 `tac2` 。分号是右结合的。

习题 1.

```
Lemma reverse_involutive: forall t,
  tree_reverse (tree_reverse t) = t.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 2.

```
Lemma size_nonneg: forall t,
  0 <= tree_size t.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 3.

```
Lemma reverse_result_Node: forall t t1 k t2,
  tree_reverse t = Node t1 k t2 ->
  t = Node (tree_reverse t2) k (tree_reverse t1).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

2 加强归纳

下面证明 `tree_reverse` 是一个单射。这个引理的 Coq 证明需要我们特别关注：真正需要归纳证明的结论是什么？如果选择对 `t1` 做结构归纳，那么究竟是归纳证明对于任意 `t2` 上述性质成立，还是归纳证明对于某“特定”的 `t2` 上述性质成立？如果我们按照之前的 Coq 证明习惯，用 `intros` 与 `induction t1` 两条指令开始证明，那就表示用归纳法证明一条关于“特定” `t2` 的性质。

```
Lemma tree_reverse_inj: forall t1 t2,
  tree_reverse t1 = tree_reverse t2 ->
  t1 = t2.
Proof.
  intros.
  induction t1 as [| t11 IHt11 v1 t12 IHt12].
  + destruct t2 as [| t21 v2 t22].
    (** 奠基步骤的证明可以通过对_[t2]_的分类讨论完成。*)
    - (** 如果_[t2]_是空树，那么结论是显然的。*)
      reflexivity.
    - (** 如果_[t2]_是非空树，那么前提_[H]_就能导出矛盾。*)
      simpl in H.
      (** 化简后，前提为：
          - H: Leaf = Node (tree_reverse t22) v (tree_reverse t21)
          这能直接推出矛盾。*)
      discriminate H.
      (** 当然，在这个证明中，由于之后的_[discriminate]_指令也会完成自动化简，前面
          的一条_[simpl]_指令其实是可以省略的。*)
  + (** 进入归纳步骤的证明时，证明已经无法继续。此时证明目标中的前提与结论为：
        - H: tree_reverse (Node t11 v1 t12) = tree_reverse t2
        - IHt11: tree_reverse t11 = tree_reverse t2 ->
                 t11 = t2
        - IHt12: tree_reverse t12 = tree_reverse t2 ->
                 t12 = t2
        - 结论: Node t11 v t12 = t2
        我们需要使用的归纳假设并非关于原_[t2]_值的性质。*)
Abort.
```

正确的证明方法是用归纳法证明一条对于一切 `t2` 的结论。

```
Lemma tree_reverse_inj: forall t1 t2,
  tree_reverse t1 = tree_reverse t2 ->
  t1 = t2.
Proof.
  intros t1.
  (** 上面这条_[intros t1]_指令可以精确地将_[t1]_放到证明目标的前提中，同时却将
      _[t2]_保留在待证明目标的结论中。*)
  induction t1 as [| t11 IHt11 v1 t12 IHt12].
  + (** 现在的奠基步骤需要证明，对于任意_[t2]_，
        - 如果_[tree_reverse Leaf = tree_reverse t2]_
        - 那么_[Leaf = t2]_ *)
    simpl. intros.
    destruct t2 as [| t21 v2 t22].
    - reflexivity.
    - discriminate H.
  + (** 现在的归纳步骤中，归纳假设为，
        - IHt11:
            forall t2: tree,
              tree_reverse t11 = tree_reverse t2 ->
              t11 = t2
        - IHt12:
            forall t2: tree,
              tree_reverse t12 = tree_reverse t2 ->
              t12 = t2 *)
    simpl. intros.
    (** 接下去对_[t2]_分类讨论，排除掉_[t2]_为空树的情况。*)
    destruct t2 as [| t21 v2 t22].
    - discriminate H.
    - injection H as H2 Hv H1.
      (** 现在，由原先的前提_[tree_reverse t1 = tree_reverse t2]_我们就知道：
          - H1: tree_reverse t11 = tree_reverse t21
          - Hv: v1 = v2
          - H2: tree_reverse t12 = tree_reverse t22
          下面就只需要使用归纳假设就可以证明_[t1 = t2]_了，即
          - Node t11 v1 t12 = Node t21 v2 t22。*)
      rewrite (IHt11 t21 H1).
      rewrite (IHt12 t22 H2).
      rewrite Hv.
      reflexivity.
Qed.
```

当然，上面这条引理其实可以不用归纳法证明。下面的证明中使用了前面证明的结论：`reverse_involutive`。

```
Lemma tree_reverse_inj_again: forall t1 t2,
  tree_reverse t1 = tree_reverse t2 ->
  t1 = t2.
Proof.
  intros.
  rewrite <- (reverse_involutive t1), <- (reverse_involutive t2).
  rewrite H.
  reflexivity.
Qed.
```

习题 4. 下面的 `left_most` 函数与 `right_most` 函数计算了二叉树中最左侧的节点信息与最右侧的节点信息。如果树为空，则返回 `default` 。

```
Fixpoint left_most (t: tree) (default: Z): Z :=
  match t with
  | Leaf => default
  | Node l n r => left_most l n
  end.
```

```
Fixpoint right_most (t: tree) (default: Z): Z :=
  match t with
  | Leaf => default
  | Node l n r => right_most r n
  end.
```

很显然，这两个函数应当满足：任意一棵二叉树的最右侧节点，就是将其左右翻转之后最左侧节点。这个性质可以在 Coq 中如下描述：

```
Lemma left_most_reverse: forall t default,
  left_most (tree_reverse t) default = right_most t default.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 5. 一棵二叉树显然不等于自己的左子树，但是这一性质应该如何在 Coq 中证明呢？

```
Theorem not_left_of_self: forall t1 v t2,
  Node t1 v t2 = t1 -> False.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

3 用递归函数定义性质

我们同样可以利用递归函数定义二叉树的一些性质。

```
Fixpoint tree_le (ub: Z) (t: tree): Prop :=
  match t with
  | Leaf => True
  | Node l k r => tree_le ub l /\ k <= ub /\ tree_le ub r
  end.
```

```
Fixpoint tree_ge (lb: Z) (t: tree): Prop :=
  match t with
  | Leaf => True
  | Node l k r => tree_ge lb l /\ k >= lb /\ tree_ge lb r
  end.
```

这里，`tree_le n t` 表示树中每个节点标号的值都小于等于 `n`，类似的，`tree_ge n t` 表示树中每个节点标号的值都大于等于 `n`。之后我们会用

`t ⊆ [n, + ∞]` 与 `t ⊆ [- ∞, n]`

表示 `tree_le n t` 与 `tree_ge n t`。

进一步，我们还可以定义，树中元素根据中序遍历是从小到大排列的 `low2high`，或从大到小排列的 `high2low` 这两条性质。

```
Fixpoint low2high (t: tree): Prop :=
  match t with
  | Leaf => True
  | Node l k r => low2high l /\ l ⊆ [- ∞, k] /\ r ⊆ [k, + ∞] /\ low2high r
  end.
```

```
Fixpoint high2low (t: tree): Prop :=
  match t with
  | Leaf => True
  | Node l k r => high2low l /\ l ⊆ [k, + ∞] /\ r ⊆ [- ∞, k] /\ high2low r
  end.
```

下面证明一些关于它们的有趣性质。我们先试着证明：如果 `t` 中元素中序遍历是从小到大的，那么将其左右翻转后，其中元素的中序遍历是从大到小的。

```
Lemma reverse_low2high: forall t,
  low2high t ->
  high2low (tree_reverse t).
Proof.
  intros.
  induction t.
  + (** 奠基步骤是显然成立的。*)
    simpl. tauto.
  + simpl in H.
    simpl.
    (** 归纳步骤的证明目标是：
        - H: low2high t1 /\ t1 ⊆ [ - ∞ , v] /\
             t2 ⊆ [v, + ∞ ] /\ low2high t2
        - IHt1: low2high t1 ->
                high2low (tree_reverse t1)
        - IHt2: low2high t2 ->
                high2low (tree_reverse t2)
        - 结论: high2low (tree_reverse t2) /\
                tree_reverse t2 ⊆ [v, + ∞ ] /\
                tree_reverse t1 ⊆ [ - ∞ , v] /\
                high2low (tree_reverse t1)
        这样看来，我们需要一些关于_[tree_le]_与_[tree_ge]_的辅助引理。*)
Abort.
```

下面首先证明，如果一棵树中的元素都小于等于 `n` ，那么它左右取反后，树中的元素依然都小于等于 `n` 。

```
Lemma reverse_le:
  forall n t,
    t ⊆ [- ∞, n] ->
    tree_reverse t ⊆ [- ∞, n].
Proof.
  intros.
  induction t; simpl.
  + tauto.
  + simpl in H.
    tauto.
Qed.
```

其次证明相对称的关于 `tree_ge` 的引理。

```
Lemma reverse_ge:
  forall n t,
    t ⊆ [n, + ∞] ->
    tree_reverse t ⊆ [n, + ∞].
Proof.
  intros.
  induction t; simpl.
  + tauto.
  + simpl in H.
    tauto.
Qed.
```

现在，准备工作已经就绪，可以开始证明 `reverse_low2high` 了。

```
Lemma reverse_low2high: forall t,
  low2high t ->
  high2low (tree_reverse t).
Proof.
  intros.
  induction t; simpl.
  + tauto.
  + simpl in H.
    pose proof reverse_le v t1.
    pose proof reverse_ge v t2.
    tauto.
Qed.
```

最后，我们再定义一个关于两棵树的性质，并证明几个基本结论。

```
Fixpoint same_structure (t1 t2: tree): Prop :=
  match t1, t2 with
  | Leaf, Leaf =>
      True
  | Leaf, Node _ _ _ =>
      False
  | Node _ _ _, Leaf =>
      False
  | Node l1 _ r1, Node l2 _ r2 =>
      same_structure l1 l2 /\ same_structure r1 r2
  end.
```

这个定义说的是：两棵二叉树结构相同，但是每个节点上标号的值未必相同。从这一定义的语法也不难看出，Coq 中允许同时对多个对象使用 `match` 并且可以用下划线省去用不到的 `match` 信息。

下面证明，如果两棵树结构相同，那么它们的高度也相同。

```
Lemma same_structure_same_height: forall t1 t2,
  same_structure t1 t2 ->
  tree_height t1 = tree_height t2.
Proof.
  intros.
  (** 要证明这一结论，很自然的思路是要对_[t1]_做结构归纳证明。这样一来，当_[t1]_
      为非空树时，归纳假设大致是：_[t1]_的左右子树分别与_[t2]_的左右子树结构相
      同，显然，这样的归纳假设可以理解推出最终的结论。*)
  induction t1.
  (** 下面先进行奠基步骤的证明。*)
  + destruct t2.
    - reflexivity.
    - simpl in H.
      tauto.
  + (** 下面进入归纳步骤。然而，通过观察此时的证明目标，我们会发现，当前证明目标与
        我们先前的设想并不一致！我们设想的证明步骤中，归纳假设应当是归于_[t2]_的子
        树的，而非归于_[t2]_本身的。这里的问题在于，当我们执行_[induction t1]_证明
        指令时，_[t2]_已经在证明目标的前提中了，这意味着，我们告诉Coq，要对某个特
        定的_[t2]_完成后续证明。这并不是我们先前拟定的证明思路。*)
Abort.
```

解决这一问题的办法是像我们先前介绍的那样采用加强归纳法。

```
Lemma same_structure_same_height: forall t1 t2,
  same_structure t1 t2 ->
  tree_height t1 = tree_height t2.
Proof.
  intros t1.
  induction t1 as [| l1 IHl v1 r1 IHr]; intros.
  + (** 奠基步骤与原先类似。 *)
    destruct t2.
    - reflexivity.
    - simpl in H.
      tauto.
  + (** 归纳步骤中，归纳假设现在不同了 *)
    destruct t2 as [| l2 v2 r2]; simpl in H.
    - tauto.
    - destruct H as [Hl Hr].
      (** 现在我们可以将归纳假设作用在_[t2]_的子树上了。 *)
      pose proof IHl l2 Hl.
      pose proof IHr r2 Hr.
      simpl.
      lia.
Qed.
```

习题 6. 下面的证明留作习题。

```
Theorem same_structure_trans: forall t1 t2 t3,
  same_structure t1 t2 ->
  same_structure t2 t3 ->
  same_structure t1 t3.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

4 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 7.

```
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 8.

```
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 9. 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 10. 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 11. 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 12. 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。 *)
```

习题 13. 从左到右遍历一颗二叉树的所有节点可以得到这棵树所有节点构成的列表。这个列表可以如下定义为 Coq 中的递归函数。

```
Fixpoint tree_elements (t: tree): list Z :=
  match t with
  | Leaf => nil
  | Node t1 v t2 => tree_elements t1 ++ v :: tree_elements t2
  end.
```

例如，下面这棵树的 `tree_elements` 是 `[3; 5; 8; 100; 9]` 。

```
      5
     / \
    3  100
       / \
      8   9
```

下面请证明 `tree_elements` 为空当且仅当二叉树为空树。

```
Theorem tree_elements_is_nil:
  forall t, tree_elements t = nil -> t = Leaf.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

下面请证明 `tree_reverse` 之后的 `tree_elements` 是原来 `tree_elements` 取反的结果。

```
Theorem tree_elements_tree_reverse:
  forall t,
    tree_elements (tree_reverse t) = rev (tree_elements t).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 14. 左旋与右旋是二叉树的常见操作。

```
        5                100
       / \               / \
      3  100           5   9
         / \          / \
        8   9        3   8
```

上面两图中，左图左旋后就会得到右图，右图右旋后又可以还原为左图，显然，这样的左旋与右旋操作并不会改变一棵树的 `tree_elements` 。这可以概括为下面性质。

```
Fact rotate_tree_elements:
  forall t1 t2 t3 x y,
    tree_elements (Node t1 x (Node t2 y t3)) =
    tree_elements (Node (Node t1 x t2) y t3).
Proof. intros. simpl. rewrite <- app_assoc. simpl. reflexivity. Qed.
```

下面定义的 Coq 函数不断地将一个二叉树右旋，直到将最左侧的节点旋转到根。

```
Fixpoint lift_leftmost_rec (t1: tree) (x: Z) (t2: tree): tree :=
  match t1 with
  | Leaf => Node t1 x t2
  | Node t11 v t12 => lift_leftmost_rec t11 v (Node t12 x t2)
  end.
```

```
Definition lift_leftmost (t: tree): tree :=
  match t with
  | Leaf => Leaf
  | Node t1 x t2 => lift_leftmost_rec t1 x t2
  end.
```

请证明该函数保持树的 `tree_elements` 不变。

```
Lemma tree_elements_lift_leftmost_rec:
  forall t1 x t2,
    tree_elements (lift_leftmost_rec t1 x t2) =
    tree_elements t1 ++ x :: tree_elements t2.
(* 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.
```

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.
```

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 15. 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 16. 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 17. 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。 *)
```

从前面的介绍中不难看出，在许多的证明场合，`list` 是左右不对称的。例如，在 `app`（`++`）的结合律证明中，我们必须对左边的列表使用归纳法，而不能对右边的列表使用归纳法。许多读者可能会好奇，既然 `list` 相关的证明中左右不对称性，与直观相背，那么能否修改定义，使其左右对称呢？从 `list` 的定义看，其左右不对称的根源在于 `cons` 构造子，其表示在已有列表的左侧添加一个元素构成一个新的列表，但 `list` 类型的定义中缺没有与之对应的表示“右侧添加元素”的构造子。下面我们试着定义一种新的 `list` 类型，使得它具有两侧对称的构造子，并分析其是否能够满足我们的要求。

下面定义的 `sym_list` 具有三个构造子：

```
Inductive sym_list (A: Type): Type :=
  | sym_nil: sym_list A
  | left_cons (a: A) (l: sym_list A): sym_list A
  | right_cons (l: sym_list A) (a: A): sym_list A.
```

```
Arguments sym_nil {A}.
Arguments left_cons {A} _ _.
Arguments right_cons {A} _ _.
```

分别是 `sym_nil` 表示空列表，`left_cons` 表示在左边添加元素，以及 `right_cons` 表示在右边添加元素。如果我们定义 `sym_list` 上的左右取反函数，那么这个取反函数本身也具有左右对称性。

```
Fixpoint sym_rev {A: Type} (l: sym_list A): sym_list A :=
  match l with
  | sym_nil => sym_nil
  | left_cons a l => right_cons (sym_rev l) a
  | right_cons l a => left_cons a (sym_rev l)
  end.
```

不能证明，`sym_rev` 具有对合性。

```
Theorem sym_rev_involutive: forall {A: Type} (l: sym_list A),
  sym_rev (sym_rev l) = l.
Proof.
  intros.
  induction l; simpl; congruence.
Qed.
```

然而，上面定义的 `sym_list` 实际上并不是一个好的列表定义方法。该类型定义首先给功能类似于 `app` 的列表连接函数的定义带来了不少麻烦。而更重要的是，该类型不具有唯一表示性，即可能多个 `sym_list` 类型的元素表示同一个列表。例如，在 `list` 类型中，`cons 1 nil` 表示长度为 1 唯一元素为 1 的列表。但是在 `sym_list` 中，

- `left_cons 1 sym_nil`
- `right_cons sym_nil 1`

根据 Coq 归纳类型的定义它们是不同的数学对象，但是我们希望用它们表示同一个长度为 1 的列表。

```
Fact sym_list_sample_neq:
  left_cons 1 sym_nil <> right_cons sym_nil 1.
Proof. congruence. Qed.
```

由此可以看出，这样左右对称的类型 `sym_list` 不能用于定义列表。

5 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 18. 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 19. 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 20. 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 21. 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。 *)
```