基于代数结构的证明支持

1 等价关系

数学上，一个二元关系“≡”是一个等价关系当且仅当它满足下面三个性质：

- 自反性：`forall a, a ≡ a`
- 对称性：`forall a b, a ≡ b -> b ≡ a`
- 传递性：`forall a b c, a ≡ b -> b ≡ c -> a ≡ c`

有很多二元关系是等价关系，例如下面定义的 `same_sgn` 说的是两个整数符号相同，即同为负、同为零或者同为正。

```
Definition same_sgn (x y: Z): Prop :=
  x < 0 /\ y < 0 \/ x = 0 /\ y = 0 \/ x > 0 /\ y > 0.
```

我们可以依次证明，`same_sgn` 具有自反性、对称性和传递性。

```
Theorem same_sgn_refl: forall x: Z,
  same_sgn x x.
Proof. unfold same_sgn. lia. Qed.
```

```
Theorem same_sgn_symm: forall x y: Z,
  same_sgn x y ->
  same_sgn y x.
Proof. unfold same_sgn. lia. Qed.
```

```
Theorem same_sgn_trans: forall x y z: Z,
  same_sgn x y ->
  same_sgn y z ->
  same_sgn x z.
Proof. unfold same_sgn. lia. Qed.
```

我们还知道，如果 `x1` 、`x2` 、`x3` 、`x4` 和 `x5` 中任意相邻两个都同号，那么它们五个全都同号。下面为了简单起见，我们先证明 `x1` 与 `x5` 同号。

```
Example same_sgn5: forall x1 x2 x3 x4 x5: Z,
  same_sgn x1 x2 ->
  same_sgn x2 x3 ->
  same_sgn x3 x4 ->
  same_sgn x4 x5 ->
  same_sgn x1 x5.
Proof.
  intros x1 x2 x3 x4 x5 H12 H23 H34 H45.
  (** 证明的主体就是反复使用传递性。*)
  pose proof same_sgn_trans x1 x2 x3 H12 H23 as H13.
  pose proof same_sgn_trans x1 x3 x4 H13 H34 as H14.
  pose proof same_sgn_trans x1 x4 x5 H14 H45 as H15.
  apply H15.
Qed.
```

下面是一个类似的证明，它要用到 `same_sgn` 的对称性和传递性。

```
Example same_sgn4: forall x1 x2 x3 x4: Z,
  same_sgn x1 x2 ->
  same_sgn x3 x2 ->
  same_sgn x3 x4 ->
  same_sgn x1 x4.
Proof.
  intros x1 x2 x3 x4 H12 H32 H34.
  pose proof same_sgn_symm x3 x2 H32 as H23.
  pose proof same_sgn_trans x1 x2 x3 H12 H23 as H13.
  pose proof same_sgn_trans x1 x3 x4 H13 H34 as H14.
  apply H14.
Qed.
```

这个证明过程也可以用反向证明替代。

```
Example same_sgn4_alternative1: forall x1 x2 x3 x4: Z,
  same_sgn x1 x2 ->
  same_sgn x3 x2 ->
  same_sgn x3 x4 ->
  same_sgn x1 x4.
Proof.
  intros x1 x2 x3 x4 H12 H32 H34.
  apply (same_sgn_trans x1 x2 x4).
  + apply H12.
  + apply (same_sgn_trans x2 x3 x4).
    - apply same_sgn_symm.
      apply H32.
    - apply H34.
Qed.
```

然而，上面这几个命题更直观的证明思路也许应当用 `rewrite` 来刻画。例如，当我们证明整数相等的类似性质时，我们可以下面这样写证明。

```
Example Zeq_ex: forall x1 x2 x3 x4: Z,
  x1 = x2 ->
  x3 = x2 ->
  x3 = x4 ->
  x1 = x4.
Proof.
  intros x1 x2 x3 x4 H12 H32 H34.
  rewrite H12, <- H32, H34.
  reflexivity.
Qed.
```

Coq 标准库提供了自反、对称、传递与等价的统一定义，并基于这些统一定义提供了 `rewrite` 、`reflexivity` 等证明指令支持。下面三条证明中，`Reflexive` 、`Symmetric` 与 `Transitive` 是 Coq 标准库对于自反、对称与传递的定义。Coq 标准库还将这三个定义注册成了 Coq 的 Class，这使得 Coq 能够提供一些特定的证明支持。这里的关键字也不再使用 `Lemma` 或 `Theorem` ，而是使用 `Instance` ，这表示 Coq 将在后续证明过程中为 `same_sgn` 提供自反、对称与传递相关的专门支持。

```
#[export] Instance same_sgn_refl': Reflexive same_sgn.
Proof. unfold Reflexive. apply same_sgn_refl. Qed.
```

```
#[export] Instance same_sgn_symm': Symmetric same_sgn.
Proof. unfold Symmetric. apply same_sgn_symm. Qed.
```

```
#[export] Instance same_sgn_trans': Transitive same_sgn.
Proof. unfold Transitive. apply same_sgn_trans. Qed.
```

Coq 还将这三条性质打包起来，定义了等价关系 `Equivalence` 。要在 Coq 中证明 `same_sgn` 是一个等价关系，可以使用 `split` 指令，将“等价关系”规约为“自反性”、“对称性”与“传递性”。

```
#[export] Instance same_sgn_equiv: Equivalence same_sgn.
Proof.
  split.
  + apply same_sgn_refl'.
  + apply same_sgn_symm'.
  + apply same_sgn_trans'.
Qed.
```

现在，我们可以用 `rewrite` 与 `reflexivity` 重新证明上面的性质：

```
Example same_sgn4_alternative2: forall t1 t2 t3 t4,
  same_sgn t1 t2 ->
  same_sgn t3 t2 ->
  same_sgn t3 t4 ->
  same_sgn t1 t4.
Proof.
  intros t1 t2 t3 t4 H12 H32 H34.
  rewrite H12, <- H32, H34.
  reflexivity.
Qed.
```

细究这个证明过程，`rewrite H12` 利用

`same_sgn t1 t2`

将 `same_sgn t1 t4` 规约为 `same_sgn t2 t4`，这其实就是使用了 `same_sgn` 的传递性！类似的，`rewrite <- H32` 使用了传递性与对称性，`rewrite H34` 又一次使用了传递性，而最后的 `reflexivity` 使用了自反性。

先前我们利用 Coq 归纳类型与递归函数定义了二叉树 `tree` 与二叉树结构相等 `same_structure`。我们还证明过，`same_structure` 具有传递性（`same_structure_trans`），事实上，我们还知道 `same_structure` 是一个等价关系！

```
Lemma same_structure_refl: forall t: tree,
  same_structure t t.
```

```
Lemma same_structure_symm: forall t1 t2: tree,
  same_structure t1 t2 -> same_structure t2 t1.
```

基于等价关系，我们可以对等价的元素进行替换。

```
Example same_structure_ex: forall t1 t2 t3 t4,
  same_structure t1 t2 ->
  same_structure t3 t2 ->
  same_structure t3 t4 ->
  same_structure t1 t4.
```

它的 Coq 证明如下：

```
Proof.
  intros t1 t2 t3 t4 H12 H32 H34.
  apply (same_structure_trans t1 t2 t4).
  + apply H12.
  + apply (same_structure_trans t2 t3 t4).
    - apply same_structure_symm.
      apply H32.
    - apply H34.
Qed.
```

这样的证明方式是不方便的。我们将等价关系的证明用 `Instance` 关键字重述。

```
#[export] Instance same_structure_refl': Reflexive same_structure.
Proof. unfold Reflexive. apply same_structure_refl. Qed.
```

```
#[export] Instance same_structure_symm': Symmetric same_structure.
Proof. unfold Symmetric. apply same_structure_symm. Qed.
```

```
#[export] Instance same_structure_trans': Transitive same_structure.
Proof. unfold Transitive. apply same_structure_trans. Qed.
```

```
#[export] Instance same_structure_equiv: Equivalence same_structure.
Proof.
  split.
  + apply same_structure_refl'.
  + apply same_structure_symm'.
  + apply same_structure_trans'.
Qed.
```

现在，我们可以用 `rewrite` 与 `reflexivity` 重新证明上面的性质：

```
Example same_structure_ex_alternative: forall t1 t2 t3 t4,
  same_structure t1 t2 ->
  same_structure t3 t2 ->
  same_structure t3 t4 ->
  same_structure t1 t4.
Proof.
  intros t1 t2 t3 t4 H12 H32 H34.
  rewrite H12, <- H32, H34.
  reflexivity.
Qed.
```

习题 1. 我们称两个整数模 5 同余当且仅当它们刚好差一个 5 的倍数。

```
Definition congr_mod5 (a b: Z): Prop :=
  exists k: Z, a - b = 5 * k.
```

下面请证明模 5 同余是一个等价关系。

```
#[export] Instance congr_mod5_refl: Reflexive congr_mod5.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
#[export] Instance congr_mod5_symm: Symmetric congr_mod5.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
#[export] Instance congr_mod5_trans: Transitive congr_mod5.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
#[export] Instance congr_mod5_equiv: Equivalence congr_mod5.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 2. 如果把一个序列（`list`）看作环状的，那么

`[1; 2; 3; 4]`

`[2; 3; 4; 1]`

`[3; 4; 1; 2]`

`[4; 1; 2; 3]`

表示的是同一个环。下面定义的 `rotate` 就表示，可以通过轮转变换由一个序列得到另一个序列，换言之，这两个序列表示的是同一个环。

```
Definition rotate {A: Type} (l1 l2: list A): Prop :=
  exists lx ly, l1 = lx ++ ly /\ l2 = ly ++ lx.
```

首先，请证明 `rotate` 具有自反性：

```
#[export] Instance rotate_refl {A: Type}: Reflexive (@rotate A).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

其次，请证明 `rotate` 具有对称性。

```
#[export] Instance rotate_symm {A: Type}: Symmetric (@rotate A).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

要证明 `rotate` 具有传递性要复杂一些。根据定义，

`rotate l1 l2`

`rotate l2 l3`

这两个条件等价于存在 `lx` 、 `ly` 、 `lu` 与 `lv` 使得：

`l1 = lx ++ ly`

`l2 = ly ++ lx`

`l2 = lu ++ lv`

`l3 = lv ++ lu`

观察中间两条性质，我们立即知道 `ly ++ lx = lu ++ v` 。这意味着，要么 `l2` 可以写成 `ly ++ l ++ lv` 的形式，并且

`lx = l ++ lv`

`lu = ly ++ l` ;

要么 `l2` 可以写成 `lu ++ l ++ lx` 的形式，并且

`ly = lu ++ l`

`lv = l ++ lx` 。

无论是两种情况中的哪一种成立，都足以帮助我们证明 `rotate l1 l3` 。下面，请首先证明这条证明传递性时需要用到的重要引理，请注意恰当选择归纳证明的对象，也要恰当选择加强归纳的方法。

```
Lemma app_split3: forall {A: Type} (lx ly lu lv: list A),
  ly ++ lx = lu ++ lv ->
  (exists l, ly = lu ++ l /\ lv = l ++ lx) \/
  (exists l, lu = ly ++ l /\ lx = l ++ lv).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

接下去，请用上面的引理 `app_split3` 证明 `rotate` 有传递性。

```
#[export] Instance rotate_trans {A: Type}: Transitive (@rotate A).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

现在我们已经证明了 `rotate` 是一个等价关系，因此就可以在 Coq 中写如下证明了。

```
Example rotate_ex: forall {A: Type} (l1 l2 l3 l4: list A),
  rotate l1 l2 ->
  rotate l3 l2 ->
  rotate l3 l4 ->
  rotate l1 l4.
Proof.
  intros A l1 l2 l3 l4 H12 H32 H34.
  rewrite H12, <- H32, H34.
  reflexivity.
Qed.
```

2 函数与等价关系

在 Coq 中，除了可以像前面那样构建归纳类型数学对象之间的等价关系之外，还可以构造函数之间的等价关系。例如，在考察 `A -> B` 类型的所有函数时，就可以基于 `B` 类型上的等价关系，利用“逐点等价”定义这些函数之间的等价关系。“逐点等价”说的是，函数 `f` 与 `g` 等价当且仅当对于任意一个定义域中的元素 `a` 都用 `f a` 与 `g a` 等价。这一定义就是 Coq 标准库中的 `pointwise_relation` 。

```
Definition pointwise_relation
             (A: Type) {B: Type}
             (R: B -> B -> Prop)
             (f g: A -> B): Prop :=
  forall a: A, R (f a) (g a).
```

Coq 标准库也证明了，如果 `R` 是等价关系，那么 `pointwise_relation A R` 也是等价关系。下面首先证明，如果 `R` 具有自反性，那么 `pointwise_relation A R` 也具有自反性。

```
#[export] Instance pointwise_reflexive:
  forall {A B: Type} {R: B -> B -> Prop},
    Reflexive R ->
    Reflexive (pointwise_relation A R).
Proof.
  intros.
  unfold Reflexive, pointwise_relation.
  (** 展开定义后需要证明
      - _[forall (x: A -> B) a, R (x a) (x a)]_。*)
  intros.
  reflexivity.
  (** 这一步是使用二元关系_[R]_的自反性完成证明。*)
Qed.
```

在上面的证明中，之所以最后可以用 `reflexivity` 指令证明 `R (x a) (x a)` 是因为在证明目标中有一条前提 `H: Reflexive R` 。事实上，Coq 对于等价关系等代数性质的支持，不仅仅限于用 `Instance` 注册过的结构，也包括在证明前提中预设的结构。此处既然假设了 `R` 具有自反性，而且自反性是使用 Coq 标准库中的 `Reflexive` 描述的，那么在证明过程中就可以使用 `reflexivity` 完成相关证明。下面是关于对称性的结论：只要 `R` 具有对称性，`pointwise_relation A R` 就有对称性。

```
#[export] Instance pointwise_symmetric:
  forall {A B: Type} {R: B -> B -> Prop},
    Symmetric R ->
    Symmetric (pointwise_relation A R).
Proof.
  intros.
  unfold Symmetric, pointwise_relation.
  intros.
  (** 展开定义后需要证明的前提和结论是：
      - H0: forall a, R (x a) (y a)
      - 结论: R (y a) (x a) *)
  symmetry.
  (** 这里的_[symmetry]_指令表示使用对称性。*)
  apply H0.
Qed.
```

```
#[export] Instance pointwise_transitive:
  forall {A B: Type} {R: B -> B -> Prop},
    Transitive R ->
    Transitive (pointwise_relation A R).
Proof.
  intros.
  unfold Transitive, pointwise_relation.
  intros.
  (** 展开定义后需要证明的前提和结论是:
      - H0: forall a, R (x a) (y a)
      - H1: forall a, R (y a) (z a)
      - 结论: R (x a) (z a) *)
  transitivity (y a).
  (** 这里, _[transitivity (y a)]_表示用“传递性”证明并选_[y a]_作为中间元素。*)
  + apply H0.
  + apply H1.
Qed.
```

下面我们把关于自反、对称与传递的这三个结论打包起来。

```
#[export] Instance pointwise_equivalence:
  forall {A B: Type} {R: B -> B -> Prop},
    Equivalence R ->
    Equivalence (pointwise_relation A R).
(* 证明详见Coq源代码。 *)
```

在 Coq 中，普通的等号 `=` 实际是一个 Notation，其背后的定义名称为 `eq` 。这是一个多态二元谓词，例如 `@eq Z` 表示“整数相等”，`@eq (list Z)` 表示“整数列表相等”。这个等号表示的“相等”自然也是一个等价关系，这一定理在 Coq 标准库中的描述如下：

```
eq_equivalence: forall {A : Type}, Equivalence (@eq A)
```

更进一步，两个类型为 `A -> B` 的函数，“它们在 `A` 类型的自变量任意取值时求值结果都相同”就可以用下面二元关系表示：

```
Definition func_equiv (A B: Type):
  (A -> B) -> (A -> B) -> Prop :=
  pointwise_relation A (@eq B).
```

我们知道，`func_equiv` 也一定是一个等价关系。

```
#[export] Instance func_equiv_equiv:
  forall A B, Equivalence (func_equiv A B).
Proof.
  intros.
  apply pointwise_equivalence.
  apply eq_equivalence.
Qed.
```

习题 3.

我们称两个整数函数是模 5 意义上等价的，当且仅当它们在每一点上的函数值都模 5 同余。例如 $f(n) = n^5$ 与 $g(n) = n$ 就是模 5 意义上等价的。

```
Example congr_mod5_sample:
  (pointwise_relation Z congr_mod5)
    (fun n => n * n * n * n * n)
    (fun n => n).
(* 证明详见Coq源代码。 *)
```

下面请利用已有结论证明：模 5 意义下的函数等价是一种等价关系。

```
Example func_equiv_sample:
  Equivalence (pointwise_relation Z congr_mod5).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

除了可以定义函数之间的等价关系之外，我们还可以反过来利用函数构造等价关系。下面这条性质就表明，可以基于一个 `A -> B` 类型的函数 `f` 以及一个 `B` 上的等价关系构造一个 `A` 上的等价关系。这一 `A` 集合上的等价关系是：`a1` 与 `a2` 等价当且仅当 `f a1` 与 `f a2` 等价。

```
Theorem equiv_in_domain:
  forall {A B: Type} (f: A -> B) (R: B -> B -> Prop),
    Equivalence R ->
    Equivalence (fun a1 a2 => R (f a1) (f a2)).
(* 证明详见Coq源代码。 *)
```

我们可以利用这个定理证明模 5 同余是一个等价关系。

```
Definition congr_Mod5 (x y: Z): Prop := x mod 5 = y mod 5.
```

```
#[export] Instance congr_Mod5_equiv:
  Equivalence congr_Mod5.
Proof.
  apply (equiv_in_domain (fun x => x mod 5)).
  apply eq_equivalence.
Qed.
```

这里的 `mod` 是 Coq 中的整数相除取余数的运算，它对应的整数相除取整运算是 Coq 中的 `/`，它计算得到的余数总是与除数同号。换言之，它满足以下性质：

```
  Z_div_mod_eq_ful: forall a b: Z, a = b * (a / b) + a mod b
  Z_mod_lt: forall a b: Z, b > 0 -> 0 <= a mod b < b
```

标准库还提供了许多有用性质

```
  Zplus_mod: forall a b n: Z, (a + b) mod n = (a mod n + b mod n) mod n
  Zminus_mod: forall a b n: Z, (a - b) mod n = (a mod n - b mod n) mod n
  Zmult_mod: forall a b n: Z, (a * b) mod n = (a mod n * b mod n) mod n
  Z_mod_plus_full: forall a b c: Z, (a + b * c) mod c = a mod c
```

更多的相关性质还请读者在需要时利用 `Search` 指令查找。

细心的读者可能已经发现，刚才定义的 `congr_Mod5` 与先前习题中定义的 `congr_mod5` 在数学上是同一个二元关系。

```
Theorem congr_mod5_Mod5: forall x y,
  congr_mod5 x y <-> congr_Mod5 x y.
```

我们可以在 Coq 中证明它们之间的等价性。

```
Proof.
  intros; unfold congr_mod5, congr_Mod5; split; intros.
  + (** 第一个分支对应以下证明目标：
          - H : exists k : Z, x - y = 5 * k
          - 结论：x mod 5 = y mod 5
        这一个分支的关键是得出
          - x = y + k * 5
        这样就可以代入后利用_[Z_mod_plus_full]_完成证明了。 *)
    destruct H as [k H].
    assert (x = y + k * 5) as Hx.
    { lia. }
    rewrite Hx.
    apply Z_mod_plus_full.
  + (** 第一个分支对应以下证明目标：
          - H : x mod 5 = y mod 5
          - 结论：exists k : Z, x - y = 5 * k
        这一个分支的关键在于论证：结论中的_[k]_就是_[x / 5 - y / 5]_。*)
    pose proof Z_div_mod_eq_full x 5.
    pose proof Z_div_mod_eq_full y 5.
    exists (x/5 - y/5).
    lia.
Qed.
```

习题 4.

假如 `RA` 与 `RB` 分别是 `A` 与 `B` 上的等价关系，那么每个 `A -> B` 的函数都可以看作 `RA` 等价类到 `RB` 等价类的对应关系，只不过，这些对应关系不一定是一一对应，而往往是从 `RA` 等价类到 `RB` 等价类的一多对应。如果 `f` 与 `g` 都是 `A -> B` 类型的函数，而且它们总是把同样的 `RA` 等价类对应到同样的 `RB` 等价类，那么我们就称它们之间具有 `lift_rel RA RB` 关系。请证明 `lift_rel RA RB` 也是一个等价关系。

```
Definition lift_rel
             {A B: Type}
             (RA: A -> A -> Prop)
             (RB: B -> B -> Prop)
             (f g: A -> B): Prop :=
  (forall a1: A, exists a2: A, RA a1 a2 /\ RB (f a1) (g a2)) /\
  (forall a1: A, exists a2: A, RA a1 a2 /\ RB (g a1) (f a2)).
```

```
#[local] Instance lift_rel_reflexive:
  forall {A B: Type} (RA: A -> A -> Prop) (RB: B -> B -> Prop),
    Reflexive RA ->
    Reflexive RB ->
    Reflexive (lift_rel RA RB).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
#[local] Instance lift_rel_symmetric:
  forall {A B: Type} (RA: A -> A -> Prop) (RB: B -> B -> Prop),
    Symmetric (lift_rel RA RB).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
#[local] Instance lift_rel_transitive:
  forall {A B: Type} (RA: A -> A -> Prop) (RB: B -> B -> Prop),
    Transitive RA ->
    Transitive RB ->
    Transitive (lift_rel RA RB).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
#[local] Instance lift_rel_equivalence:
  forall {A B: Type} (RA: A -> A -> Prop) (RB: B -> B -> Prop),
    Equivalence RA ->
    Equivalence RB ->
    Equivalence (lift_rel RA RB).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

3 Coq 中的 Morphisms

前面已经提到，“除以 5 同余”是数学上很有用的一个等价关系。例如，我们可以证明，对于任意一个整数 n，如果 $n+2$ 除以 5 的余数是 1，那么 n 除以 5 的余数是 4。证明如下：

$$n = (n+2) - 2 \equiv 1 - 2 = -1 \equiv 4$$

其中第一步和第三步是普通的整数运算性质，第二步和第四步是除以 5 同余的性质。

下面我们试着在 Coq 中写出这个证明。

```
Fact n_plus_2_equiv_1: forall n: Z,
  congr_Mod5 (n + 2) 1 ->
  congr_Mod5 n 4.
Proof.
  intros.
  assert (n = n + 2 - 2).
  { lia. }
  (** 现在我们已经做好了预先准备，现在的前提有：
      - H: congr_Mod5 (n + 2) 1
      - H0: n = n + 2 - 2
    接下去，按照之前的计划，我们只需要依次利用这两个等式重写就好了。 *)
  rewrite H0.
  (** 现在待证结论是：
      - congr_Mod5 (n + 2 - 2) 4 *)
  Fail rewrite H.
  (** 但是Coq拒绝了这里的第二条_[rewrite]_。 *)
Abort.
```

仔细检查这一步骤，我们会发现，要将

```
congr_Mod5 (n + 2 - 2) 4
```

规约为 `congr_Mod5 (1 - 2) 4` ，除了需要使用前提 `congr_Mod5 (n + 2) 1` 还需要用到“减法保持模 5 同余”。换言之，因为

```
congr_Mod5 (n + 2) 1
```

```
congr_Mod5 2 2
```

所以， `congr_Mod5 (n + 2 - 2) (1 - 2)` 。下面我们先证明“减法保持模 5 同余”。

```
Lemma Zsub_preserves_congr_Mod5: forall x1 x2 y1 y2,
  congr_Mod5 x1 x2 ->
  congr_Mod5 y1 y2 ->
  congr_Mod5 (x1 - y1) (x2 - y2).
Proof.
  unfold congr_Mod5; intros.
  (** 根据_[congr_Mod5]_的定义，现在只需证明：
        - H: x1 mod 5 = x2 mod 5
        - H0: y1 mod 5 = y2 mod 5
        - 结论：(x1 - y1) mod 5 = (x2 - y2) mod 5 *)
  rewrite (Zminus_mod x1 y1).
  rewrite (Zminus_mod x2 y2).
    (** 使用_[Zminus_mod]_之后，结论被规约为：
        - (x1 mod 5 - y1 mod 5) mod 5 =
          (x2 mod 5 - y2 mod 5) mod 5
       这样一来就可以由两个前提直接推出这个等式的左右两边相等了。 *)
  rewrite H, H0.
  reflexivity.
Qed.
```

下面我们再试着重新在 Coq 中证明 `n_plus_2_equiv_1` 。

```
Fact n_plus_2_equiv_1_attempt: forall n: Z,
  congr_Mod5 (n + 2) 1 ->
  congr_Mod5 n 4.
Proof.
  intros.
  assert (n = n + 2 - 2).
  { lia. }
  rewrite H0.
  (** 现在待证结论是：
      - congr_Mod5 (n + 2 - 2) 4
    注意，此时依然不能直接使用_[rewrite H]_，读者可以自行尝试。 *)
  assert (congr_Mod5 (n + 2 - 2) (1 - 2)). {
    apply Zsub_preserves_congr_Mod5.
    + apply H.
    + reflexivity.
  }
  rewrite H1.
  (** 现在只需证明_[congr_Mod5 (1 - 2) 4]_。 *)
  reflexivity.
Qed.
```

在上面这段证明中，我们成功利用“减法保持模 5 同余”构造了

```
congr_Mod5 (n + 2 - 2) (1 - 2)
```

的证明。但是由于不能直接使用 `rewrite` ，这个 Coq 证明方法依然不够简明。而究其原因，关键在于 Coq 无法基于引理

```
Zsub_preserves_congr_Mod5
```

的表述获知其中关键的代数结构。先前我们所使用的 Coq 中基于代数结构的证明指令都搭建在 `Reflexive` 、 `Symmetric` 、 `Transitive` 与 `Equivalence` 等专门定义的基础之上。Coq 标准库其实也提供了“保持等价性”的 Coq 定义，这就是 `Proper` 。下面，我们利用 `Proper` 重述“减法保持模 5 同余”这一性质。

```
#[export] Instance Proper_congr_Mod5_Zsub:
  Proper (congr_Mod5 ==> congr_Mod5 ==> congr_Mod5) Z.sub.
```

这条性质中的 `Z.sub` 就是整数减法，当我们在 Coq 表达式中写整数类型的表达式 `x - y` 时，实际的意思就是 `Z.sub x y` 。上面这条性质说的是：`Z.sub` 是一个二元函数，如果对其两个参数分别做 `congr_Mod5` 变换，那么这个二元函数的计算结果也发生 `congr_Mod5` 变换。在证明这一结论时，需要展开 `Proper` 的定义，还需要展开 `==>` 的定义，它的 Coq 名字是 `respectful` 。

```
Proof.
  unfold Proper, respectful, congr_Mod5; intros.
  (** 展开_[Proper]_等定义后，需要证明的目标是：
        - H: x mod 5 = y mod 5
        - H0: x0 mod 5 = y0 mod 5
        - 结论: (x - x0) mod 5 = (y - y0) mod 5 *)
  rewrite (Zminus_mod x x0).
  rewrite (Zminus_mod y y0).
  rewrite H, H0.
  reflexivity.
Qed.
```

下面我们重新证明 `n_plus_2_equiv_1` 。

```
Fact n_plus_2_equiv_1: forall n: Z,
  congr_Mod5 (n + 2) 1 ->
  congr_Mod5 n 4.
Proof.
  intros.
  assert (n = n + 2 - 2).
  { lia. }
  rewrite H0.
  rewrite H. (** 这一行就用到了前面证明的_[Proper]_性质。*)
  reflexivity.
Qed.
```

我们还可以在 Coq 中证明整数的加法和乘法也会保持“模 5 同余”。

```
#[export] Instance Proper_congr_Mod5_Zadd:
  Proper (congr_Mod5 ==> congr_Mod5 ==> congr_Mod5) Z.add.
```

```
#[export] Instance Proper_congr_Mod5_Zmul:
  Proper (congr_Mod5 ==> congr_Mod5 ==> congr_Mod5) Z.mul.
```

下面是另一个关于被 5 除同余的简单证明。

```
Fact n_mult_3_equiv_2: forall n: Z,
  congr_Mod5 (n * 3) 2 ->
  congr_Mod5 n 4.
Proof.
  intros.
  assert (n = n * 1).
  { lia. }
  assert (congr_Mod5 1 6).
  { reflexivity. }
  assert (n * 6 = n * 3 * 2).
  { lia. }
  (** 当前的证明目标是：
        - H: congr_Mod5 (n * 3) 2
        - H0: n = n * 1
        - H1: congr_Mod5 1 6
        - H2: n * 6 = n * 3 * 2
        - 结论: congr_Mod5 n 4
      这只需要依次_[rewrite]_就可以完成证明了。*)
  rewrite H0, H1, H2, H.
  reflexivity.
Qed.
```

下面我们看几个关于 `same_structure` 的代数性质：许多树变换都能保持树结构的等价关系。例如：如果

`same_structure t11 t12`

`same_structure t21 t22`

那么 `Node t11 n1 t21` 与 `Node t12 n2 t22` 也是结构相同的。运用我们现在所学，这个性质可以写成下面引理。

```
Lemma Node_same_structure_congr: forall t11 t12 t21 t22 n1 n2,
  same_structure t11 t12 ->
  same_structure t21 t22 ->
  same_structure (Node t11 n1 t21) (Node t12 n2 t22).
(* 证明详见Coq源代码。 *)
```

类似地，`tree_reverse` 也能保持树结构地等价关系。只不过这一引理需要用归纳法证明。归纳证明中的奠基步骤是要证明

`same_structure (tree_reverse Leaf) (tree_reverse Leaf)`

这是显然的。证明中的归纳步骤是要基于

`same_structure (tree_reverse t11) (tree_reverse t21)`

`same_structure (tree_reverse t12) (tree_reverse t22)`

这两条归纳假设推出：

`same_structure`
`(Node (tree_reverse t11) n1 (tree_reverse t12))`
`(Node (tree_reverse t21) n2 (tree_reverse t22))` 。

我们现在已经知道 `same_structure` 是等价关系，也知道 `Node` 可以保持这一等价关系，因此我们希望可以像证明等式相关性质那样使用 `rewrite` 指令来完成这一证明。然而 Coq 并不支持我们现在这么做。

```
Example tree_reverse_same_structure_congr_ind_step:
  forall t11 t12 t21 t22 n1 n2,
    same_structure (tree_reverse t11) (tree_reverse t21) ->
    same_structure (tree_reverse t12) (tree_reverse t22) ->
    same_structure
      (Node (tree_reverse t11) n1 (tree_reverse t12))
      (Node (tree_reverse t21) n2 (tree_reverse t22)).
Proof.
  intros.
  Fail rewrite H, H0.
Abort.
```

下面是 `same_structure` 与等号 `=` 的对比：

```
Example same_structure_vs_eq:
  forall t11 t12 t21 t22 n,
    tree_reverse t11 = tree_reverse t21 ->
    tree_reverse t12 = tree_reverse t22 ->
    Node (tree_reverse t11) n (tree_reverse t12) =
    Node (tree_reverse t21) n (tree_reverse t22).
Proof.
  intros.
  rewrite H, H0.
  reflexivity.
Qed.
```

之所以 Coq 目前无法支持对 `same_structure` 如上面所示的那样利用 `rewrite` 指令重写，主要原因在于 Coq 无法基于引理

`Node_same_structure_congr`

的表述获知其中关键的代数结构。因此就需要使用 `Proper` 来解决这一问题。

```
Definition any {A: Type} (a b: A): Prop := True.
```

```
#[export] Instance Node_same_structure_morphism:
  Proper (same_structure ==>
          any ==>
          same_structure ==>
          same_structure) Node.
```

这个性质说得是：`Node` 是一个三元函数，如果对其第一个参数做 `same_structure` 变换，对其第二个参数做任意变换，同时对其第三个参数做 `same_structure` 变换，那么这个三元函数的计算结果也会做 `same_structure` 变换。在证明这一结论时，需要展开 `Proper` 的定义，还需要展开 `==>` 的定义，它的 Coq 名字是 `respectful`。

```
Proof.
  intros.
  unfold Proper, respectful.
  intros t11 t21 ? n1 n2 _ t12 t22 ?.
  simpl.
  tauto.
Qed.
```

下面补充证明，`any` 是一个自反关系。

```
#[export] Instance any_refl: forall A: Type, Reflexive (@any A).
(* 证明详见Coq源代码。 *)
```

这样我们就可以让 `rewrite` 用上 `Node` 保持 `same_structure` 变换这一性质了。

```
Example tree_reverse_same_structure_congr_ind_step:
  forall t11 t12 t21 t22 n1 n2,
    same_structure (tree_reverse t11) (tree_reverse t21) ->
    same_structure (tree_reverse t12) (tree_reverse t22) ->
    same_structure
      (Node (tree_reverse t11) n1 (tree_reverse t12))
      (Node (tree_reverse t21) n2 (tree_reverse t22)).
Proof.
  intros.
  rewrite H, H0.
  simpl; split; reflexivity.
Qed.
```

自然，`tree_reverse` 保持 `same_structure` 也可以用 `Proper` 刻画。

```
#[export] Instance tree_reverse_same_structure_morphism:
  Proper (same_structure ==> same_structure) tree_reverse.
```

上面的例子中用 `Proper` 描述了 `Node` 与 `tree_reverse` 这两个函数的性质。其实 `Proper` 也可以用于描述谓词的性质。例如，下面性质说的是，如果对 `same_structure` 这个谓词的两个参数分别做 `same_structure` 变换，那么变换前后的两个命题要么全都成立要么全都不成立，即变换之前的命题成立当且仅当变换之后的命题成立，这个“当且仅当”就是下面定理描述中的 `iff` 。

```
#[export] Instance same_structure_same_structure_morphism:
  Proper (same_structure ==> same_structure ==> iff) same_structure.
Proof.
  unfold Proper, respectful.
  intros.
  rewrite H, H0.
  reflexivity.
Qed.
```

下面定义的 `structural_subtree` 描述了结构上的子树关系（不考虑节点上的数值）。

```
Fixpoint structural_subtree (t1 t2: tree): Prop :=
  match t2 with
  | Leaf =>
      same_structure t1 t2
  | Node t21 n2 t22 =>
      same_structure t1 t2 \/
      structural_subtree t1 t21 \/
      structural_subtree t1 t22
  end.
```

很显然，对 `structural_subtree` 的参数做 `same_structure` 不改变命题的真假。

```
#[export] Instance structural_subtree_same_structure_morphism:
  Proper (same_structure ==> same_structure ==> iff) structural_subtree.
(* 证明详见Coq源代码。 *)
```

然而,如果对 `structural_subtree` 的两个参数都做 `structural_subtree` 变换,并不能确保变换前后 `structural_subtree` 性质不变。具体而言，如果已知

`structural_subtree t1 t2`

并对 `t1` 与 `t2` 做如下变换

`structural_subtree t1 t1'`

`structural_subtree t2 t2'`

并不能得出

`structural_subtree t1' t2'`

这一结论。不过，如果修正从 `t1` 到 `t1'` 的变换方向，`structural_subtree` 这一性质就能被保持。具体而言，我们可以由：

`structural_subtree t1 t2`

`structural_subtree t1' t1`

`structural_subtree t2 t2'`

推出

`structural_subtree t1' t2'` 。

这一性质可以用 `Proper` 概述为：

```
#[export] Instance structural_subtree_structural_subtree_morphism:
  Proper
    (structural_subtree -->
     structural_subtree ==>
     Basics.impl) structural_subtree.
```

其中，`-->` 表示第一个参数做 `structural_subtree` 的反向变换，“反向”这一概念对应的 Coq 定义是 `Basics.flip`；另外，`Basics.impl` 表示变换前的命题可以推出变换后的命题。

```
Proof.
  unfold Proper, respectful, Basics.flip, Basics.impl.
  intros t1 t1' Ht1 t2 t2' Ht2 H.
  (** 证明指令运行至此，可以需要证明的证明目标是
        - Ht1: structural_subtree t1' t1
        - Ht2: structural_subtree t2 t2'
        - H: structural_subtree t1 t2
        - 结论: structural_subtree t1' t2'
      即是我们前述的性质。*)
```

不过，要证明这一结论，比较好的方法是先证明 `structural_subtree` 具有传递性，再连续使用两次传递性完成证明。相关的证明留作习题。

习题 5. 请证明 `structural_subtree` 具有传递性。提示：我们已经证明过，做 `same_structure` 变换能保持 `structural_subtree` 性质，因此，可以使用 `rewrite` 表述相关证明。

```
#[export] Instance structural_subtree_trans: Transitive structural_subtree.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

接下去，请运用传递性证明下面性质。

```
#[export] Instance structural_subtree_structural_subtree_morphism:
  Proper
    (structural_subtree -->
     structural_subtree ==>
     Basics.impl) structural_subtree.
Proof.
  unfold Proper, respectful, Basics.flip, Basics.impl.
  intros t1 t1' Ht1 t2 t2' Ht2 H.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

不难看出，具有传递性的二元关系都具有这样的性质。换言之，如果 `R` 是一个二元关系，那么只要 `Transitive R` 成立，就有

```
Proper (R --> R ==> Basics.impl) R
```

成立。类似的，只要 `Equivalence R` 成立，就有

```
Proper (R ==> R ==> iff) R
```

成立。这也是 Coq 能够依据等价关系 `Equivalence` 性质提供 `rewrite` 支持的原因。

习题 6.

下面定义的 `lift_leftmost` 是将二叉树的最左侧节点旋转到根。

```
Fixpoint lift_leftmost_aux (t1: tree) (n: Z) (t2: tree): tree :=
  match t1 with
  | Leaf => Node Leaf n t2
  | Node t11 n1 t12 => lift_leftmost_aux t11 n1 (Node t12 n t2)
  end.
```

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

请证明它能保持 `same_structure` 。

```
#[export] Instance lift_leftmost_aux_same_structure_morphism:
  Proper (same_structure ==>
          any ==>
          same_structure ==>
          same_structure) lift_leftmost_aux.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
#[export] Instance lift_leftmost_same_structure_morphism:
  Proper (same_structure ==>
          same_structure) lift_leftmost.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

4 匿名函数内部的重写

考虑对整数列表的操作：对列表中的每一项都增加一个常数。这一操作可以使用 `list` 有关的标准库函数 `map` 定义。

```
Definition Zlist_inc (x: Z) (l: list Z): list Z :=
  map (fun x0 => x0 + x) l.
```

自然，`Zlist_inc` 应当满足下面两条基本性质：

`Zlist_inc 0 l = l`

`Zlist_inc x (Zlist_inc y l) = Zlist_inc (x + y) l`

我们尽管可以使用 Coq 中的结构归纳法证明这两个结论，但我们更希望能利用 Coq 中 `map` 函数的性质与单个算数运算的性质证明它们。

```
Theorem Zlist_inc_0:
  forall l, Zlist_inc 0 l = l.
Proof.
  intros.
  unfold Zlist_inc.
  (** 这时需要证明_[map (fun x0 => x0 + 0) l = l]_。要证明这一结论，很自然的想法是利
      用_[Z.add_0_r: forall z, z + 0 = z]_将_[map]_的第一个参数变为自反函数
      _[fun x0 => x0]_。然而此时并不能直接_[rewrite]_。*)
  Fail rewrite Z.add_0_r.
  (** 如果_[rewrite]_，Coq会给出类似“Found no subterm matching "? + 0" in the
      current goal.”的报错信息。其原因是，待证明结论中的_[x0 + 0]_单独看其本身不是一
      个合法的项，它其中出现的_[x0]_受到_[fun x0 => ...]_匿名函数的约束。*)
Abort.
```

要在带约束变量的情况下重写，需要先证明相应的 `Proper` 性质（往往与 `pointwise_relation` 有关），再使用 `setoid_rewrite` 指令重写。例如，证明 `Zlist_inc_0` 需要用到的是 `map_eq_proper` 。它说的是，如果 `map` 的第一个参数变换为相同的函数（即两个函数在自变量在定义域内任意取值的情况下求值结果都相同），那么 `map` 的计算结果相同。

```
#[export] Instance map_eq_proper {A B: Type}:
  Proper (pointwise_relation _ eq ==> eq ==> eq) (@map A B).
```

下面重新证明 `Zlist_inc_0` 。

```
Theorem Zlist_inc_0:
  forall l, Zlist_inc 0 l = l.
Proof.
  intros.
  unfold Zlist_inc.
  setoid_rewrite Z.add_0_r.
  (** _[setoid_rewrite]_指令成功的将_[map]_的第一个参数变为了自反函数，即当前只需要
      证明_[map (fun x0 => x0) l = l]_即可，这就是_[map_id]_。*)
  apply map_id.
Qed.
```

下面 `Zlist_inc_Zlist_inc` 的证明也是类似的。

```
Theorem Zlist_inc_Zlist_inc:
  forall x y l,
    Zlist_inc x (Zlist_inc y l) = Zlist_inc (x + y) l.
Proof.
  intros.
  unfold Zlist_inc.
  rewrite map_map.
  (** 现在需要证明：
        _[map (fun x0 => x0 + y + x) l = map (fun x0 => x0 + (x + y) l)]_
      这就需要在带约束变量的环境内使用加法结合律，和加法交换律。*)
  setoid_rewrite <- Z.add_assoc.
  (** 值得一提的是，要证明_[fun x0 => x0 + (y + x) = fun x0 => x0 + (x + y)]_只需
      要使用普通的_[rewrite]_即可，因为_[x + y]_与_[y + x]_中不包含约束变量_[x0]_*)
  rewrite Z.add_comm.
  reflexivity.
Qed.
```

习题 7. 先前我们曾定义 `suffixes` 函数用于计算一个列表的所有后缀。这个 `suffixes` 是通过从右向左的计算方式定义的。下面定义一种从左向右计算的版本。请你证明它与 `suffixes` 的结果永远相同。

```
Fixpoint suffixes_L2R_rec
           {A: Type}
           (l: list A)
           (res: list (list A)): list (list A) :=
  match l with
  | nil => res ++ [nil]
  | a :: l0 => suffixes_L2R_rec
                 l0
                 (map (fun suf => suf ++ [a]) res ++ [[a]])
  end.
```

```
Definition suffixes_L2R {A: Type} (l: list A): list (list A) :=
  suffixes_L2R_rec l [].
```

如果需要，你可以写出并证明一些辅助引理帮助你完成下面定理的证明。

```
Theorem suffixes_L2R_suffixes:
  forall A (l: list A),
    suffixes_L2R l = suffixes l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 8. 请证明下面定义的从左向右计算的 `prefixes_L2R` 与先前定义的从右向左计算的 `prefixes` 是计算结果相同的函数。

```
Fixpoint prefixes_L2R_rec
           {A: Type}
           (l res1: list A)
           (res2: list (list A)): list (list A) :=
  match l with
  | nil => res2 ++ [res1]
  | a :: l0 => prefixes_L2R_rec
                 l0
                 (res1 ++ [a])
                 (res2 ++ [res1])
  end.
```

```
Definition prefixes_L2R {A: Type} (l: list A): list (list A) :=
  prefixes_L2R_rec l [] [].
```

如果需要，你可以写出并证明一些辅助引理帮助你完成下面定理的证明。

```
Theorem prefixes_L2R_prefixes:
  forall A (l: list A), prefixes_L2R l = prefixes l.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```