Coq 中的代数结构

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。

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.
```

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。

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.
```