课后阅读：在 Coq 中证明 Kleene 不动点定理

1 Coq 中证明并应用 Kleene 不动点定理

```
Module KleeneFix.
```

下面我们将在 Coq 中证明 Kleene 不动点定理。在 Kleene 不动点定理中，我们需要证明满足某些特定条件（例如偏序、完备偏序等）的二元关系的一些性质。在 Coq 中，我们当然可以通过 `R: A -> A -> Prop` 来探讨二元关系 `R` 的性质。然而 Coq 中并不能给这样的变量设定 Notation 符号，例如，我们希望用 `a <= b` 来表示 `R a b` ，因此我们选择使用 Coq 的 `Class` 来帮助我们完成定义。

下面这一定义说的是：`Order` 是一类数学对象，任给一个类型 `A` ，`Order A` 也是一个类型，这个类型的每个元素都有一个域，这个域的名称是 `order_rel` ，它的类型是 `A -> A -> Prop` ，即 `A` 上的二元关系。

```
Class Order (A: Type): Type :=
  order_rel: A -> A -> Prop.
```

Coq 中 `Class` 与 `Record` 有些像，但是有两点区别。第一：`Class` 如果只有一个域，它的中可以不使用大括号将这个域的定义围起来；第二：在定义或证明中，Coq 系统会试图自动搜索并填充类型为 `Class` 的参数，搜索范围为之前注册过可以使用的 `Instance` 以及当前环境中的参数。例如，我们先前在证明等价关系、congruence 性质时就使用过 `Instance` 。例如，下面例子中，不需要指明 `order_rel` 是哪个 `Order A` 的 `order_rel` 域，Coq 会自动默认这是指 `RA` 的 `order_rel` 域。

```
Check forall {A: Type} {RA: Order A} (x: A),
        exists (y: A), order_rel x y.
```

这样，我们就可以为 `order_rel` 定义 Notation 符号。

```
Declare Scope order_scope.
Notation "a <= b" := (order_rel a b): order_scope.
Local Open Scope order_scope.

Check forall {A: Type} {RA: Order A} (x y: A),
        x <= y \/ y <= x.
```

基于序关系，我们就可以定义上界与下界的概念。由于 Kleene 不动点定理中主要需要探讨无穷长元素序列的上界与上确界，下面既定义了元素集合的上下界也定义了元素序列的上下界。

```
Definition is_lb
             {A: Type} {RA: Order A}
             (X: A -> Prop) (a: A): Prop :=
  forall a', X a' -> a <= a'.
```

```
Definition is_ub
             {A: Type} {RA: Order A}
             (X: A -> Prop) (a: A): Prop :=
  forall a', X a' -> a' <= a.

Definition is_omega_lb
             {A: Type} {RA: Order A}
             (l: nat -> A) (a: A): Prop :=
  forall n, a <= l n.
```

```
Definition is_omega_ub
             {A: Type} {RA: Order A}
             (l: nat -> A) (a: A): Prop :=
  forall n, l n <= a.
```

下面定义序列的上确界，所谓上确界就是上界中最小的一个，因此它的定义包含两个子句。而在后续证明中，使用上确界性质的时候，有时需要用其第一条性质，有时需要用其第二条性质。为了后续证明的方便，这里在定义之外提供了使用这两条性质的方法：`is_omega_lub_sound` 与 `is_omega_lub_tight` 。比起在证明中使用 `destruct` 指令拆解上确界的定义，使用这两条引理，可以使得 Coq 证明更自然地表达我们的证明思路。之后我们将在证明偏序集上上确界唯一性的时候会看到相关的用法。

```
Definition is_omega_lub
             {A: Type} {RA: Order A}
             (l: nat -> A) (a: A): Prop :=
  is_omega_ub l a /\ is_lb (is_omega_ub l) a.

Lemma is_omega_lub_sound:
  forall {A: Type} {RA: Order A} {l: nat -> A} {a: A},
    is_omega_lub l a -> is_omega_ub l a.
Proof. unfold is_omega_lub; intros; tauto. Qed.

Lemma is_omega_lub_tight:
  forall {A: Type} {RA: Order A} {l: nat -> A} {a: A},
    is_omega_lub l a -> is_lb (is_omega_ub l) a.
Proof. unfold is_omega_lub; intros; tauto. Qed.
```

在编写 Coq 定义时，另有一个问题需要专门考虑，有些数学上的相等关系在 Coq 中只是一种等价关系。例如，我们之前在 Coq 中用过集合相等的定义。因此，我们描述 Kleene 不动点定理的前提条件时，也需要假设有一个与序关系相关的等价关系，我们用 `Equiv` 表示，并用 `==` 这个符号描述这个等价关系。

```
Class Equiv (A: Type): Type :=
  equiv: A -> A -> Prop.
```

```
Notation "a == b" := (equiv a b): order_scope.
```

基于此，我们可以定义基于等价关系的自反性与反对称性。注意，传递性的定义与这个等价关系无关。这里我们也用 `Class` 定义，这与 Coq 标准库中的自反、对称、传递的定义是类似的，但是也有不同：(1) 我们的定义需要探讨一个二元关系与一个等价关系之间的联系，而 Coq 标准库中只考虑了这个等价关系是普通等号的情况；(2) Coq 标准库中直接使用二元关系 `R: A -> A -> Prop` 作为参数，而我们的参数使用了 `Order` 与 `Equiv` 这两个 `Class` 。

自反性：

```
Class Reflexive_Setoid
        (A: Type) {RA: Order A} {EA: Equiv A}: Prop :=
  reflexivity_setoid:
    forall a b, a == b -> a <= b.
```

反对称性：

```
Class AntiSymmetric_Setoid
        (A: Type) {RA: Order A} {EA: Equiv A}: Prop :=
  antisymmetricity_setoid:
    forall a b, a <= b -> b <= a -> a == b.
```

现在，我们就可以如下定义偏序关系。

```
Class PartialOrder_Setoid
        (A: Type) {RA: Order A} {EA: Equiv A}: Prop :=
{
  PO_Reflexive_Setoid:: Reflexive_Setoid A;
  PO_Transitive:: Transitive order_rel;
  PO_AntiSymmetric_Setoid:: AntiSymmetric_Setoid A
}.
```

下面证明两条偏序集的基本性质。在 Coq 中，我们使用前引号 `` ` `` 让 Coq 自动填充 `Class` 类型元素的参数。例如，`` `{POA: PartialOrder_Setoid A} `` 会指引 Coq 额外填上 `RA: Order A` 和 `EA: Equiv A` 。

序关系两侧做等价变换不改变序关系：

```
#[export] Instance PartialOrder_Setoid_Proper
          {A: Type} `{POA: PartialOrder_Setoid A} {EquivA: Equivalence equiv}:
  Proper (equiv ==> equiv ==> iff) order_rel.
(* 证明详见Coq源代码。 *)
```

如果两个序列的所有上界都相同，那么他们的上确界也相同（如果有的话）：

```
Lemma same_omega_ub_same_omega_lub:
  forall
    {A: Type}
    `{POA: PartialOrder_Setoid A}
    (l1 l2: nat -> A)
    (a1 a2: A),
  (forall a, is_omega_ub l1 a <-> is_omega_ub l2 a) ->
  is_omega_lub l1 a1 ->
  is_omega_lub l2 a2 ->
  a1 == a2.
(* 证明详见Coq源代码。 *)
```

证明 Kleene 不动点定理时还需要定义完备偏序集，由于在其证明中实际只用到了完备偏序集有最小元和任意单调不减的元素序列有上确界，我们在 Coq 定义时也只考虑符合这两个条件的偏序集，我们称为 OmegaCPO，Omega 表示可数无穷多项的意思。另外，尽管数学上仅仅要求完备偏序集上的所有链有上确界，但是为了 Coq 证明的方便，我们将 `omega_lub` 定义为所有元素序列的上确界计算函数，只不过我们仅仅要求该函数在其参数为单调不减序列时能确实计算出上确界，见 `oCPO_completeness` 。

```
Class OmegaLub (A: Type): Type :=
  omega_lub: (nat -> A) -> A.
```

```
Class Bot (A: Type): Type :=
  bot: A.

Definition increasing
             {A: Type} {RA: Order A} (l: nat -> A): Prop :=
  forall n, l n <= l (S n).
```

```
Definition is_least {A: Type} {RA: Order A} (a: A): Prop :=
  forall a', a <= a'.
```

```
Class OmegaCompletePartialOrder_Setoid
        (A: Type)
        {RA: Order A} {EA: Equiv A}
        {oLubA: OmegaLub A} {BotA: Bot A}: Prop :=
{
  oCPO_PartialOrder:: PartialOrder_Setoid A;
  oCPO_completeness: forall T,
    increasing T -> is_omega_lub T (omega_lub T);
  bot_is_least: is_least bot
}.
```

利用这里定义中的 `omega_lub` 函数，可以重述先前证明过的性质：两个单调不减序列如果拥有完全相同的上界，那么他们也有同样的上确界。

```
Lemma same_omega_ub_same_omega_lub':
  forall
    {A: Type}
    `{oCPOA: OmegaCompletePartialOrder_Setoid A}
    (l1 l2: nat -> A),
  (forall a, is_omega_ub l1 a <-> is_omega_ub l2 a) ->
  increasing l1 ->
  increasing l2 ->
  omega_lub l1 == omega_lub l2.
(* 证明详见Coq源代码。 *)
```

下面定义单调连续函数。

```
Definition mono
             {A B: Type}
             `{POA: PartialOrder_Setoid A}
             `{POB: PartialOrder_Setoid B}
             (f: A -> B): Prop :=
  forall a1 a2, a1 <= a2 -> f a1 <= f a2.

Definition continuous
             {A B: Type}
             `{oCPOA: OmegaCompletePartialOrder_Setoid A}
             `{oCPOB: OmegaCompletePartialOrder_Setoid B}
             (f: A -> B): Prop :=
  forall l: nat -> A,
    increasing l ->
    f (omega_lub l) == omega_lub (fun n => f (l n)).
```

下面我们可以证明：自反函数是单调连续的、复合函数能保持单调连续性。

自反函数的单调性：

```
Lemma id_mono:
  forall {A: Type}
         `{POA: PartialOrder_Setoid A},
  mono (fun x => x).
(* 证明详见Coq源代码。 *)
```

复合函数保持单调性：

```
Lemma compose_mono:
  forall {A B C: Type}
         `{POA: PartialOrder_Setoid A}
         `{POB: PartialOrder_Setoid B}
         `{POC: PartialOrder_Setoid C}
         (f: A -> B)
         (g: B -> C),
  mono f -> mono g -> mono (fun x => g (f x)).
(* 证明详见Coq源代码。 *)
```

自反函数的连续性：

```
Lemma id_continuous:
  forall {A: Type}
         `{oCPOA: OmegaCompletePartialOrder_Setoid A}
         {EquivA: Equivalence equiv},
  continuous (fun x => x).
(* 证明详见Coq源代码。 *)
```

这里，要证明单调连续函数的复合结果也是连续的要复杂一些。显然，这其中需要证明一个单调函数作用在一个单调不减序列的每一项后还会得到一个单调不减序列。下面的引理 `increasing_mono_increasing` 描述了这一性质。

```
Lemma increasing_mono_increasing:
  forall {A B: Type}
         `{POA: PartialOrder_Setoid A}
         `{POB: PartialOrder_Setoid B}
         (f: A -> B)
         (l: nat -> A),
  increasing l -> mono f -> increasing (fun n => f (l n)).
(* 证明详见Coq源代码。 *)
```

除此之外，我们还需要证明单调函数能保持相等关系，即，如果 `f` 是一个单调函数，那么 `x == y` 能推出 `f x == f y` 。当然，如果这里的等价关系就是等号描述的相等关系，那么这个性质是显然的。但是，对于一般的等价关系，这就并不显然了。这一引理的正确性依赖于偏序关系中的自反性和反对称性。

```
Lemma mono_equiv_congr:
  forall {A B: Type}
         `{POA: PartialOrder_Setoid A}
         `{POB: PartialOrder_Setoid B}
          {EquivA: Equivalence (equiv: A -> A -> Prop)}
         (f: A -> B),
  mono f -> Proper (equiv ==> equiv) f.
(* 证明详见Coq源代码。 *)
```

现在，可以利用上面两条引理证明复合函数的连续性了。

```
Lemma compose_continuous:
  forall {A B C: Type}
         `{oCPOA: OmegaCompletePartialOrder_Setoid A}
         `{oCPOB: OmegaCompletePartialOrder_Setoid B}
         `{oCPOC: OmegaCompletePartialOrder_Setoid C}
          {EquivB: Equivalence (equiv: B -> B -> Prop)}
          {EquivC: Equivalence (equiv: C -> C -> Prop)}
         (f: A -> B)
         (g: B -> C),
  mono f ->
  mono g ->
  continuous f ->
  continuous g ->
  continuous (fun x => g (f x)).
(* 证明详见Coq源代码。 *)
```

到目前为止，我们已经定义了 Omega 完备偏序集与单调连续函数。在证明 Kleene 不动点定理之前还需要最后一项准备工作：定理描述本身的合法性。即，我们需要证明 `bot` , `f bot` , `f (f bot)` ... 这个序列的单调性。我们利用 Coq 标准库中的 `Nat.iter` 来定义这个序列，`Nat.iter n f bot` 表示将 `f` 连续 `n` 次作用在 `bot` 上。

```
Lemma iter_bot_increasing:
  forall
    {A: Type}
    `{oCPOA: OmegaCompletePartialOrder_Setoid A}
    (f: A -> A),
    mono f ->
    increasing (fun n => Nat.iter n f bot).
(* 证明详见Coq源代码。 *)
```

当然，`f bot` , `f (f bot)` , `f (f (f bot))` ... 这个序列也是单调不减的。

```
Lemma iter_S_bot_increasing:
  forall
    {A: Type}
    `{oCPOA: OmegaCompletePartialOrder_Setoid A}
    (f: A -> A),
    mono f ->
    increasing (fun n => f (Nat.iter n f bot)).
(* 证明详见Coq源代码。 *)
```

`Kleene_LFix` 定义了 Kleene 最小不动点。

```
Definition Kleene_LFix
             {A: Type}
             `{CPOA: OmegaCompletePartialOrder_Setoid A}
             (f: A -> A): A :=
  omega_lub (fun n => Nat.iter n f bot).
```

先证明，`Kleene_LFix` 的计算结果确实是一个不动点。

```
Lemma Kleene_LFix_is_fix:
  forall
    {A: Type}
    `{CPOA: OmegaCompletePartialOrder_Setoid A}
    {EquivA: Equivalence equiv}
    (f: A -> A),
    mono f ->
    continuous f ->
    f (Kleene_LFix f) == Kleene_LFix f.
(* 证明详见Coq源代码。 *)
```

再证明，`Kleene_LFix` 的计算结果是最小不动点。

```
Lemma Kleene_LFix_is_least_fix:
  forall
    {A: Type}
    `{CPOA: OmegaCompletePartialOrder_Setoid A}
    {EquivA: Equivalence equiv}
    (f: A -> A)
    (a: A),
    mono f ->
    continuous f ->
    f a == a ->
    Kleene_LFix f <= a.
(* 证明详见Coq源代码。 *)


End KleeneFix.
```

接下去我们将利用 Kleene 最小不动点定义 While 语句的程序语义中运行终止的情况。

首先需要定义我们所需的 OmegaCPO。在定义 `Class` 类型的值时，可以使用 `Intance` 关键字。如果 `Class` 中只有一个域并且 `Class` 的定义没有使用大括号包围所有域，那么这个域的定义就是整个 `Class` 类型的值的定义；否则 `Class` 类型的值应当像 `Record` 类型的值一样定义。

```
#[export] Instance R_while_fin {A B: Type}: Order (A -> B -> Prop) :=
  Sets.included.
```

```
#[export] Instance Equiv_while_fin {A B: Type}: Equiv (A -> B -> Prop) :=
  Sets.equiv.
```

下面证明这是一个偏序关系。证明的时候需要展开上面两个二元关系（一个表示序关系另一个表示等价关系）。以序关系为例，此时需要将 `R_while_fin` 与 `order_rel` 全部展开，前者表示将上面的定义展开，后者表示将从 `Class Order` 取出 `order_rel` 域这一操作展开。其余的证明则只需用 `Sets_unfold` 证明集合相关的性质。

```
#[export] Instance PO_while_fin {A B: Type}: PartialOrder_Setoid (A -> B -> Prop).
(* 证明详见Coq源代码。 *)
```

下面再定义上确界计算函数与完备偏序集中的最小值。

```
#[export] Instance oLub_while_fin {A B: Type}: OmegaLub (A -> B -> Prop) :=
  Sets.indexed_union.
```

```
#[export] Instance Bot_while_fin {A B: Type}: Bot (A -> B -> Prop) :=
  ∅: A -> B -> Prop.
```

下面证明这构成一个 Omega 完备偏序集。

```
#[export] Instance oCPO_while_fin {A B: Type}:
  OmegaCompletePartialOrder_Setoid (A -> B -> Prop).
(* 证明详见Coq源代码。 *)
```

额外需要补充一点：`Equiv_while_fin` 确实是一个等价关系。先前 Kleene 不动点定理的证明中用到了这一前提。

```
#[export] Instance Equiv_equiv_while_fin {A B: Type}:
  Equivalence (@equiv (A -> B -> Prop) _).
Proof.
  apply Sets_equiv_equiv.
Qed.
```

有时，Coq 证明中需要展开上面这些定理，这可以使用下面的证明指令。

```
Ltac unfold_CPO_defs :=
  unfold order_rel, equiv, omega_lub, bot,
         R_while_fin, Equiv_while_fin, oLub_while_fin, Bot_while_fin.
```

下面开始证明 `F(X) = (test_true(D0) ∘ D ∘ X) ∪ test_false(D0)` 这个函数的单调性与连续性。整体证明思路是：(1) `F(X) = X` 是单调连续的；(2) 如果 `F` 是单调连续的，那么 `G(X) = Y ∘ F(X)` 也是单调连续的；(3) 如果 `F` 是单调连续的，那么 `G(X) = F(X) ∪ Y` 也是单调连续的；其中 `Y` 是给定的二元关系。

下面证明前面提到的步骤 (2)：如果 `F` 是单调连续的，那么 `G(X) = Y ∘ F(X)` 也是单调连续的。主结论为 `BinRel_concat_left_mono_and_continuous` ，其用到了两条下面的辅助引理以及前面证明过的复合函数单调连续性定理。

```
Lemma BinRel_concat_left_mono:
  forall (A B C: Type) (Y: A -> B -> Prop),
    mono (fun X: B -> C -> Prop => Y ∘ X).
(* 证明详见Coq源代码。 *)
```

```
Lemma BinRel_concat_left_continuous:
  forall (A B C: Type) (Y: A -> B -> Prop),
    continuous (fun X: B -> C -> Prop => Y ∘ X).
(* 证明详见Coq源代码。 *)
```

```
Lemma BinRel_concat_left_mono_and_continuous:
  forall
    (A B C: Type)
    (Y: A -> B -> Prop)
    (f: (B -> C -> Prop) -> (B -> C -> Prop)),
  mono f /\ continuous f ->
  mono (fun X => Y ∘ f X) /\ continuous (fun X => Y ∘ f X).
(* 证明详见Coq源代码。 *)
```

下面证明前面提到的步骤 (3)：如果 `F` 是单调连续的，那么 `G(X) = Y ∘ F(X)` 也是单调连续的。主结论为 `union_right2_mono_and_continuous` ，其用到了两条下面的辅助引理以及前面证明过的复合函数单调连续性定

理。

```
Lemma union_right2_mono:
  forall (A B: Type) (Y: A -> B -> Prop),
    mono (fun X => X ∪ Y).
(* 证明详见Coq源代码。 *)
```

```
Lemma union_right2_continuous:
  forall (A B: Type) (Y: A -> B -> Prop),
    continuous (fun X => X ∪ Y).
(* 证明详见Coq源代码。 *)
```

```
Lemma union_right2_mono_and_continuous:
  forall
    (A B: Type)
    (Y: A -> B -> Prop)
    (f: (A -> B -> Prop) -> (A -> B -> Prop)),
  mono f /\ continuous f ->
  mono (fun X => f X ∪ Y) /\ continuous (fun X => f X ∪ Y).
(* 证明详见Coq源代码。 *)
```

最终我们可以用 Kleene 不动点定义 while 语句运行终止的情况。

首先给出语义定义。

```
Definition while_sem
             (D0: state -> Prop)
             (D: state -> state -> Prop):
  state -> state -> Prop :=
  Kleene_LFix (fun X => (test_true(D0) ∘ D ∘ X) ∪ test_false(D0)).
```

下面可以直接使用 Kleene 不动点定理证明上述定义就是我们要的最小不动点。

```
Theorem while_sem_is_least_fix: forall D0 D,
  (test_true(D0) ∘ D ∘ while_sem D0 D) ∪ test_false(D0) == while_sem D0 D /\
  (forall X,
    (test_true(D0) ∘ D ∘ X) ∪ test_false(D0) == X -> while_sem D0 D ⊆ X).
(* 证明详见Coq源代码。 *)
```