指称语义

1 简单表达式的指称语义

指称语义是一种定义程序行为的方式。在极简的 SimpleWhile 语言中，整数类型表达式中只有整数常量、变量、加法、减法与乘法运算。

```
EI :: = N | V | EI + EI | EI - EI | EI * EI
```

我们约定其中整数变量的值、整数运算的结果都是没有范围限制的。基于这一约定，我们可以如下定义表达式`e`在程序状态`st`上的值。

首先定义程序状态集合：

$$\text{state} \triangleq \text{var\_name} \to \mathbb{Z}$$

```
Definition state: Type := var_name -> Z.
```

- $[\![n]\!](s) = n$
- $[\![x]\!](s) = s(x)$
- $[\![e_1 + e_2]\!](s) = [\![e_1]\!](s) + [\![e_2]\!](s)$
- $[\![e_1 - e_2]\!](s) = [\![e_1]\!](s) - [\![e_2]\!](s)$
- $[\![e_1 * e_2]\!](s) = [\![e_1]\!](s) * [\![e_2]\!](s)$

其中 $s \in \text{state}$。

下面使用 Coq 递归函数定义整数类型表达式的行为。

```
Fixpoint eval_expr_int (e: expr_int) (s: state) : Z :=
  match e with
  | EConst n => n
  | EVar X => s X
  | EAdd e1 e2 => eval_expr_int e1 s + eval_expr_int e2 s
  | ESub e1 e2 => eval_expr_int e1 s - eval_expr_int e2 s
  | EMul e1 e2 => eval_expr_int e1 s * eval_expr_int e2 s
  end.
```

下面是两个具体的例子。

```
Example eval_example1: forall (s: state),
  s "x" = 1 ->
  s "y" = 2 ->
  eval_expr_int [["x" + "y"]] s = 3.
Proof. intros. simpl. rewrite H, H0. reflexivity. Qed.
```

```
Example eval_example2: forall (s: state),
  s "x" = 1 ->
  s "y" = 2 ->
  eval_expr_int [["x" * "y" + 1]] s = 3.
Proof. intros. simpl. rewrite H, H0. reflexivity. Qed.
```

2 行为等价

基于整数类型表达式的语义定义`eval_expr_int`，我们可以定义整数类型表达式之间的行为等价（亦称语义等价）：两个表达式`e1`与`e2`是等价的当且仅当它们在任何程序状态上的求值结果都相同。

$$e_1 \equiv e_2 \quad \text{iff.} \quad \forall s.\ [\![e_1]\!]\,(s) = [\![e_2]\!]\,(s)$$

```
Definition expr_int_equiv (e1 e2: expr_int): Prop :=
  forall st, eval_expr_int e1 st = eval_expr_int e2 st.
```

```
Notation "e1 '~=~' e2" := (expr_int_equiv e1 e2)
  (at level 69, no associativity).
```

下面是一些表达式语义等价的例子。

Example 1. $x * 2 \equiv x + x$

证明. 对于任意程序状态 s, $[\![x * 2]\!]\,(s) = s(x) * 2 = s(x) + s(x) = [\![x + x]\!]\,(s)$。 □

```
Example expr_int_equiv_sample:
  [["x" + "x"]] ~=~ [["x" * 2]].
```

```
Proof.
  intros.
  unfold expr_int_equiv.
```

上面的`unfold`指令表示展开一项定义，一般用于非递归的定义。

```
  intros.
  simpl.
  lia.
Qed.

Lemma zero_plus_equiv: forall (a: expr_int),
  [[0 + a]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma plus_zero_equiv: forall (a: expr_int),
  [[a + 0]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma minus_zero_equiv: forall (a: expr_int),
  [[a - 0]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma zero_mult_equiv: forall (a: expr_int),
  [[0 * a]] ~=~ 0.
(* 证明详见Coq源代码。 *)

Lemma mult_zero_equiv: forall (a: expr_int),
  [[a * 0]] ~=~ 0.
(* 证明详见Coq源代码。 *)

Lemma const_plus_const: forall n m: Z,
  [[EConst n + EConst m]] ~=~ EConst (n + m).
(* 证明详见Coq源代码。 *)

Lemma const_minus_const: forall n m: Z,
  [[EConst n - EConst m]] ~=~ EConst (n - m).
(* 证明详见Coq源代码。 *)

Lemma const_mult_const: forall n m: Z,
  [[EConst n * EConst m]] ~=~ EConst (n * m).
(* 证明详见Coq源代码。 *)
```

下面定义一种简单的语法变换—常量折叠—并证明其保持语义等价性。所谓常量折叠指的是将只包含常量而不包含变量的表达式替换成为这个表达式的值。

```
Fixpoint fold_constants (e : expr_int) : expr_int :=
  match e with
  | EConst n    => EConst n
  | EVar x      => EVar x
  | EAdd e1 e2  =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 + n2)
      | _, _ => EAdd (fold_constants e1) (fold_constants e2)
      end
  | ESub e1 e2 =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 - n2)
      | _, _ => ESub (fold_constants e1) (fold_constants e2)
    end
  | EMul e1 e2 =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 * n2)
      | _, _ => EMul (fold_constants e1) (fold_constants e2)
    end
  end.
```

这里我们可以看到，Coq 中`match`的使用是非常灵活的。(1) 我们不仅可以对一个变量的值做分类讨论，还可以对一个复杂的 Coq 式子的取值做分类讨论；(2) 我们可以对多个值同时做分类讨论；(3) 我们可以用下划线表示`match`的缺省情况。下面是两个例子：

```
Example fold_constants_ex1:
    fold_constants [[(1 + 2) * "k"]] = [[3 * "k"]].
Proof. intros. reflexivity. Qed.
```

注意，根据我们的定义，`fold_constants`并不会将`0 + "y"`中的`0`消去。

```
Example fold_expr_int_ex2 :
  fold_constants [["x" - ((0 * 6) + "y")]] = [["x" - (0 + "y")]].
Proof. intros. reflexivity. Qed.
```

下面我们在 Coq 中证明，`fold_constants`保持表达式行为不变。

```
Theorem fold_constants_sound : forall a,
  fold_constants a ~=~ a.
Proof.
  unfold expr_int_equiv. intros.
  induction a.
```

常量的情况

```
  + simpl.
    reflexivity.
```

变量的情况

```
  + simpl.
    reflexivity.
```

加号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
```

减号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
```

乘号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
Qed.
```

3 利用高阶函数定义指称语义

```
Definition add_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s + D2 s.
```

```
Definition sub_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s - D2 s.
```

```
Definition mul_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s * D2 s.
```

下面是用于类型查询的`Check`指令。

```
Check add_sem.
```

可以看到`add_sem`的类型是`(state -> Z) -> (state -> Z) -> state -> Z`，这既可以被看做一个三元函数，也可以被看做一个二元函数，即函数之间的二元函数。

基于上面高阶函数，可以重新定义表达式的指称语义。

```
Definition const_sem (n: Z) (s: state): Z := n.
Definition var_sem (X: var_name) (s: state): Z := s X.
```

```
Fixpoint eval_expr_int (e: expr_int): state -> Z :=
  match e with
  | EConst n =>
      const_sem n
  | EVar X =>
      var_sem X
  | EAdd e1 e2 =>
      add_sem (eval_expr_int e1) (eval_expr_int e2)
  | ESub e1 e2 =>
      sub_sem (eval_expr_int e1) (eval_expr_int e2)
  | EMul e1 e2 =>
      mul_sem (eval_expr_int e1) (eval_expr_int e2)
  end.
```

4 布尔表达式语义

对于任意布尔表达式 e，我们规定它的语义 $[\![e]\!]$ 是一个程序状态到真值的函数，表示表达式 e 在各个程序状态上的求值结果。

- $[\![\mathrm{TRUE}]\!](s) = \mathbf{T}$
- $[\![\mathrm{FALSE}]\!](s) = \mathbf{F}$
- $[\![e_1 < e_2]\!](s)$ 为真当且仅当 $[\![e_1]\!](s) < [\![e_2]\!](s)$
- $[\![e_1 \&\& e_2]\!](s) = [\![e_1]\!](s)$ and $[\![e_2]\!](s)$
- $[\![!e_1]\!](s) = \mathrm{not}\ [\![e_1]\!](s)$

在 Coq 中可以如下定义：

```
Definition true_sem (s: state): bool := true.
```

```
Definition false_sem (s: state): bool := false.
```

```
Definition lt_sem (D1 D2: state -> Z) s: bool :=
  Z.ltb (D1 s) (D2 s).
```

```
Definition and_sem (D1 D2: state -> bool) s: bool :=
  andb (D1 s) (D2 s).
```

```
Definition not_sem (D: state -> bool) s: bool :=
  negb (D s).
```

```
Fixpoint eval_expr_bool (e: expr_bool): state -> bool :=
  match e with
  | ETrue =>
      true_sem
  | EFalse =>
      false_sem
  | ELt e1 e2 =>
      lt_sem (eval_expr_int e1) (eval_expr_int e2)
  | EAnd e1 e2 =>
      and_sem (eval_expr_bool e1) (eval_expr_bool e2)
  | ENot e1 =>
      not_sem (eval_expr_bool e1)
  end.
```

5 程序语句的语义

$(s_1, s_2) \in [\![c]\!]$ 当且仅当从 s_1 状态开始执行程序 c 会以程序状态 s_2 终止。

5.1 赋值语句的语义

$$[\![x = e]\!] = \{(s_1, s_2) \mid s_2(x) = [\![e]\!](s_1), \text{for any } y \in \text{var\_name},\ \text{if } x \neq y,\ s_1(y) = s_2(y)\}$$

5.2 空语句的语义

$$[\![\text{SKIP}]\!] = \{(s_1, s_2) \mid s_1 = s_2\}$$

5.3 顺序执行语句的语义

$$[\![c_1; c_2]\!] = [\![c_1]\!] \circ [\![c_2]\!] = \{(s_1, s_3) \mid (s_1, s_2) \in [\![c_1]\!], (s_2, s_3) \in [\![c_2]\!]\}$$

5.4 条件分支语句的语义

$$\begin{aligned}[\![\text{if } (e) \text{ then}\{c_1\} \text{ else } \{c_2\}]\!] = \quad & \left(\{(s_1, s_2) \mid [\![e]\!](s_1) = \mathbf{T}\} \cap [\![c_1]\!]\right) \cup \\ & \left(\{(s_1, s_2) \mid [\![e]\!](s_1) = \mathbf{F}\} \cap [\![c_2]\!]\right)\end{aligned}$$

这又可以改写为：

$[\![\text{if } (e) \text{ then}\{c_1\} \text{ else } \{c_2\}]\!] = \text{test\_true}([\![e]\!]) \circ [\![c_1]\!] \cup \text{test\_false}([\![e]\!]) \circ [\![c_2]\!]$

其中，

$$\text{test\_true}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{T}, s_1 = s_2\}$$

$$\text{test\_false}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{F}, s_1 = s_2\}$$

5.5 循环语句的语义

定义方式一：

$\text{iterLB}_0(X, R) = \text{test\_false}(X)$

$\text{iterLB}_{n+1}(X, R) = \text{test\_true}(X) \circ R \circ \text{iterLB}_n(X, R)$

$[\![\text{while } (e) \text{ do } \{c\}]\!] = \bigcup_{n\in\mathbb{N}} \text{iterLB}_n([\![e]\!], [\![c]\!])$

定义方式二：

$\text{boundedLB}_0(X, R) = \varnothing$

$\text{boundedLB}_{n+1}(X, R) = \text{test\_true}(X) \circ R \circ \text{boundedLB}_n(X, R) \cup \text{test\_false}(X)$

$[\![\text{while } (e) \text{ do } \{c\}]\!] = \bigcup_{n\in\mathbb{N}} \text{boundedLB}_n([\![e]\!], [\![c]\!])$

6 Coq 中的集合与关系

在 Coq 中往往使用 `X: A -> Prop` 来表示某类型 `A` 中元素构成的集合 `X`。字面上看，这里的 `A -> Prop` 表示 `X` 是一个从 `A` 中元素到命题的映射，这也相当于说 `X` 是一个关于 `A` 中元素性质。对于每个 `A` 中元素 `a` 而言，`a` 符合该性质 `X` 等价于 `a` 对应的命题 `X a` 为真，又等价于 `a` 是集合 `X` 的元素，在 SetsClass 库中也直接写作 `a ∈ X`。

类似的，`R: A -> B -> Prop` 也用来表示 `A` 与 `B` 中元素之间的二元关系。

本课程提供的 SetsClass 库中提供了有关集合的一系列定义。例如：

- 空集：用 `∅` 或者一堆方括号表示，定义为 `Sets.empty`；
- 单元集：用一对方括号表示，定义为 `Sets.singleton`；
- 并集：用 `∪` 表示，定义为 `Sets.union`；
- 交集：用 `∩` 表示，定义为 `Sets.intersect`；
- 一列集合的并：用 `⋃` 表示，定义为 `Sets.indexed_union`；
- 一列集合的交：用 `⋂` 表示，定义为 `Sets.indexed_intersect`；
- 集合相等：用 `==` 表示，定义为 `Sets.equiv`；
- 元素与集合关系：用 `∈` 表示，定义为 `Sets.In`；
- 子集关系：用 `⊆` 表示，定义为 `Sets.included`；
- 二元关系的连接：用 `∘` 表示，定义为 `Rels.concat`；
- 等同关系：定义为 `Rels.id`；
- 测试关系：定义为 `Rels.test`。

在 CoqIDE 中，你可以利用 CoqIDE 对于 unicode 的支持打出特殊字符：

- 首先，在打出特殊字符的 latex 表示法；
- 再按 shift+ 空格键；
- latex 表示法就自动转化为了相应的特殊字符。

例如，如果你需要打出符号 `∈`，请先在输入框中输入 `\in`，当光标紧跟在 `n` 这个字符之后的时候，按 shift+ 空格键即可。例如，下面是两个关于集合的命题：

```
Check forall A (X: A -> Prop), X ∪ ∅ == X.

Check forall A B (X Y: A -> B -> Prop), X ∪ Y ⊆ X.
```

由于集合以及集合间的运算是基于 Coq 中的命题进行定义的，集合相关性质的证明也可以规约为与命题有关的逻辑证明。例如，我们想要证明，交集运算具有交换律：

```
Lemma Sets_intersect_comm: forall A (X Y: A -> Prop),
  X ∩ Y == Y ∩ X.
Proof.
  intros.
```

下面一条命令`Sets_unfold`是 SetsClass 库提供的自动证明指令，它可以将有关集合的性质转化为有关命题的性质。

```
  Sets_unfold.
```

原本要证明的关于交集的性质现在就转化为了：`forall a : A, a ∈ X /\ a ∈ Y <-> a ∈ Y /\ a ∈ X` 这个关于逻辑的命题在 Coq 中是容易证明的。

```
  intros.
  tauto.
Qed.
```

下面是一条关于并集运算的性质。

```
Lemma Sets_included_union1: forall A (X Y: A -> Prop),
  X ⊆ X ∪ Y.
Proof.
  intros.
  Sets_unfold.
```

经过转化，要证明的结论是：`forall a : A, a ∈ X -> a ∈ X \/ a ∈ Y`。

```
  intros.
  tauto.
Qed.
```

习题 1.

下面是一条关于二元关系复合的性质。转化后得到的命题要复杂一些，请在 Coq 中证明这个关于逻辑的命题。

```
Lemma Rels_concat_assoc: forall A (X Y Z: A -> A -> Prop),
  (X ∘ Y) ∘ Z == X ∘ (Y ∘ Z).
Proof.
  intros.
  Sets_unfold.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

SetsClass 库中提供了一系列有关集合运算的性质的证明。未来大家在证明中既可以使用`Sets_unfold`将关于集合运算的命题转化为关于逻辑的命题，也可以直接使用下面这些性质完成证明。

```
Sets_equiv_Sets_included:
  forall x y, x == y <-> x ⊆ y /\ y ⊆ x;
Sets_empty_included:
  forall x, ∅ ⊆ x;
Sets_included_full:
  forall x, x ⊆ Sets.full;
Sets_intersect_included1:
  forall x y, x ∩ y ⊆ x;
Sets_intersect_included2:
  forall x y, x ∩ y ⊆ y;
Sets_included_intersect:
  forall x y z, x ⊆ y -> x ⊆ z -> x ⊆ y ∩ z;
Sets_included_union1:
  forall x y, x ⊆ x ∪ y;
Sets_included_union2:
  forall x y, y ⊆ x ∪ y;
Sets_union_included_strong2:
  forall x y z u,
     x ∩ u ⊆ z -> y ∩ u ⊆ z -> (x ∪ y) ∩ u ⊆ z;
```

```
Sets_included_indexed_union:
  forall xs n, xs n ⊆ ⋃ xs;
Sets_indexed_union_included:
  forall xs y, (forall n, xs n ⊆ y) -> ⋃ xs ⊆ y;
Sets_indexed_intersect_included:
  forall xs n, ⋂ xs ⊆ xs n;
Sets_included_indexed_intersect:
  forall xs y, (forall n : nat, y ⊆ xs n) -> y ⊆ ⋂ xs;
```

```
Rels_concat_union_distr_r:
  forall x1 x2 y,
    (x1 ∪ x2) ∘ y == (x1 ∘ y) ∪ (x2 ∘ y);
Rels_concat_union_distr_l:
  forall x y1 y2,
    x ∘ (y1 ∪ y2) == (x ∘ y1) ∪ (x ∘ y2);
Rels_concat_mono:
  forall x1 x2,
    x1 ⊆ x2 ->
    forall y1 y2,
      y1 ⊆ y2 ->
      x1 ∘ y1 ⊆ x2 ∘ y2;
Rels_concat_assoc:
  forall x y z,
    (x ∘ y) ∘ z == x ∘ (y ∘ z);
Rels_concat_id_l:
  forall x, Rels.id ∘ x == x;
Rels_concat_id_r:
  forall x, x ∘ Rels.id == x;
```

由于上面提到的集合与关系运算都能保持集合相等，也都能保持集合包含关系，因此 SetsClass 库支持其用户使用`rewrite`指令处理集合之间的相等关系与包含关系。下面是两个典型的例子

```
Fact sets_ex1:
  forall (A: Type) (R1 R2 R3 R4: A -> A -> Prop),
    R1 ∘ (R2 ∘ (R3 ∘ R4)) == ((R1 ∘ R2) ∘ R3) ∘ R4.
Proof.
  intros.
  rewrite Rels_concat_assoc.
  rewrite Rels_concat_assoc.
  reflexivity.
Qed.
```

```
Fact sets_ex2:
  forall (A: Type) (R1 R2 R3 R4: A -> A -> Prop),
    R1 ⊆ R2 ->
    R1 ∘ R3 ∪ R4 ⊆ R2 ∘ R3 ∪ R4.
Proof.
  intros.
  rewrite H.
  reflexivity.
Qed.
```

7 在 Coq 中定义程序语句的语义

下面在 Coq 中写出程序语句的指称语义。

```
Definition skip_sem: state -> state -> Prop :=
  Rels.id.
```

```
Definition asgn_sem
             (X: var_name)
             (D: state -> Z)
             (st1 st2: state): Prop :=
  st2 X = D st1 /\
  forall Y, X <> Y -> st2 Y = st1 Y.
```

```
Definition seq_sem
             (D1 D2: state -> state -> Prop):
  state -> state -> Prop :=
  D1 ∘ D2.
```

```
Definition test_true
             (D: state -> bool):
  state -> state -> Prop :=
  Rels.test (fun st => D st = true).
```

```
Definition test_false
             (D: state -> bool):
  state -> state -> Prop :=
  Rels.test (fun st => D st = false).
```

```
Definition if_sem
             (D0: state -> bool)
             (D1 D2: state -> state -> Prop):
  state -> state -> Prop :=
  (test_true D0 ∘ D1) ∪ (test_false D0 ∘ D2).
```

```
Fixpoint iterLB
           (D0: state -> bool)
           (D1: state -> state -> Prop)
           (n: nat):
  state -> state -> Prop :=
  match n with
  | O => test_false D0
  | S n0 => test_true D0 ∘ D1 ∘ iterLB D0 D1 n0
  end.
```

```
Definition while_sem
             (D0: state -> bool)
             (D1: state -> state -> Prop):
  state -> state -> Prop :=
  ⋃ (iterLB D0 D1).
```

```
Fixpoint boundedLB
           (D0: state -> bool)
           (D1: state -> state -> Prop)
           (n: nat):
  state -> state -> Prop :=
  match n with
  | O => ∅
  | S n0 =>
      (test_true D0 ∘ D1 ∘ boundedLB D0 D1 n0) ∪
      (test_false D0)
  end.
```

```
Definition while_sem
             (D0: state -> bool)
             (D1: state -> state -> Prop):
  state -> state -> Prop :=
  ⋃ (boundedLB D0 D1).
```

下面是程序语句指称语义的递归定义。

```
Fixpoint eval_com (c: com): state -> state -> Prop :=
  match c with
  | CSkip =>
      skip_sem
  | CAsgn X e =>
      asgn_sem X (eval_expr_int e)
  | CSeq c1 c2 =>
      seq_sem (eval_com c1) (eval_com c2)
  | CIf e c1 c2 =>
      if_sem (eval_expr_bool e) (eval_com c1) (eval_com c2)
  | CWhile e c1 =>
      while_sem (eval_expr_bool e) (eval_com c1)
  end.
```

基于上面定义，可以证明一些简单的程序性质。

```
Example inc_x_fact: forall s1 s2 n,
  (s1, s2) ∈ eval_com (CAsgn "x" [["x" + 1]]) ->
  s1 "x" = n ->
  s2 "x" = n + 1.
Proof.
  intros.
  simpl in H.
  unfold asgn_sem, add_sem, var_sem, const_sem in H.
  lia.
Qed.
```

更多关于程序行为的有用性质可以使用集合与关系的运算性质完成证明，`seq_skip`与`skip_seq`表明了删除顺序执行中多余的空语句不改变程序行为。

```
Lemma seq_skip: forall c,
  eval_com (CSeq c CSkip) == eval_com c.
Proof.
  intros.
  simpl.
  unfold seq_sem, skip_sem.
  apply Rels_concat_id_r.
Qed.
```

```
Lemma skip_seq: forall c,
  eval_com (CSeq CSkip c) == eval_com c.
Proof.
  intros.
  simpl.
  unfold seq_sem, skip_sem.
  apply Rels_concat_id_l.
Qed.
```

类似的，`seq_assoc`表明顺序执行的结合顺序是不影响程序行为的，因此，所有实际的编程中都不需要在程序开发的过程中额外标明顺序执行的结合方式。

```
Lemma seq_assoc: forall c1 c2 c3,
  eval_com (CSeq (CSeq c1 c2) c3) ==
  eval_com (CSeq c1 (CSeq c2 c3)).
Proof.
  intros.
  simpl.
  unfold seq_sem.
  apply Rels_concat_assoc.
Qed.
```

下面的`while_sem_congr2`说的则是：如果对循环体做行为等价变换，那么整个循环的行为也不变。

```
Lemma while_sem_congr2: forall D1 D2 D2',
  D2 == D2' ->
  while_sem D1 D2 == while_sem D1 D2'.
Proof.
  intros.
  unfold while_sem.
  apply Sets_indexed_union_congr.
  intros n.
  induction n; simpl.
  + reflexivity.
  + rewrite IHn.
    rewrite H.
    reflexivity.
Qed.
```

下面我们证明，我们先前定义的`remove_skip`变换保持程序行为不变。

```
Theorem remove_skip_sound: forall c,
  eval_com (remove_skip c) == eval_com c.
(* 证明详见Coq源代码。 *)
```

8 不动点分析

前面提到，while 循环语句的行为也可以描述为：只要循环条件成立，就先执行循环体再重新执行循环。我们可以证明，我们目前定义的程序语义符合这一性质。

- 定理：如果 $[\![\text{while } (e) \text{ do } \{c\}]\!] = \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ [\![\text{while } (e) \text{ do } \{c\}]\!] \cup \text{test\_false}([\![e]\!])$
- 证明：

$$
\begin{aligned}
& [\![\text{while } (e) \text{ do } \{c\}]\!] \\
= & \bigcup_{n\in\mathbb{N}} \text{boundedLB}_n([\![e]\!], [\![c]\!]) \\
= & \bigcup_{n\in\mathbb{N}} \text{boundedLB}_{n+1}([\![e]\!], [\![c]\!]) \\
= & \bigcup_{n\in\mathbb{N}} \Big(\text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ \text{boundedLB}_n([\![e]\!], [\![c]\!]) \cup \text{test\_false}([\![e]\!])\Big) \\
= & \ \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ \bigcup_{n\in\mathbb{N}} \text{boundedLB}_n([\![e]\!], [\![c]\!]) \cup \text{test\_false}([\![e]\!]) \\
= & \ \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ [\![\text{while } (e) \text{ do } \{c\}]\!] \cup \text{test\_false}([\![e]\!])
\end{aligned}
$$

Coq 证明可以在 Coq 源代码中找到。

从这一结论可以看出，$[\![\text{while } (e) \text{ do } \{c\}]\!]$ 是下面方程的一个解：

$$X = \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ X \cup \text{test\_false}([\![e]\!])$$

数学上，如果一个函数 f 与某个取值 x 满足 $f(x) = x$，那么就称 x 是 f 的一个不动点。因此，我们也可以说 $[\![\text{while } (e) \text{ do } \{c\}]\!]$ 是下面函数的一个不动点：

$$F(X) \triangleq \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ X \cup \text{test\_false}([\![e]\!])$$

下面分析该函数不动点的唯一性。

- 第一种情况，假设 $[\![e]\!](s_0) = \mathbf{F}$，那么

$$\text{test\_true}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{T}, s_1 = s_2\}$$

$$\text{test\_false}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{F}, s_1 = s_2\}$$

- 对于任意程序状态 s, $(s_0, s) \in \text{test\_false}([\![e]\!])$ 当且仅当 $s = s_0$；
- 不存在这样的程序状态 s 满足 $(s_0, s) \in \text{test\_true}([\![e]\!])$；

因此，对于任意一个 F 的不动点 X 以及任意程序状态 s 都有：

$(s_0, s) \in X$ 当且仅当 $(s_0, s) \in F(X)$
当且仅当 $(s_0, s) \in \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ X \cup \text{test\_false}([\![e]\!])$
当且仅当 $s = s_0$。

- 第二种情况，假设 $[\![e]\!](s_0) = \mathbf{T}$，$(s_0, s_1) \in [\![c]\!]$ 并且 $[\![e]\!](s_1) = \mathbf{F}$，那么
  - 不存在这样的程序状态 s 满足 $(s_0, s) \in \text{test\_false}([\![e]\!])$；
  - 对于任意程序状态 s, $(s_0, s) \in \text{test\_true}([\![e]\!])$ 当且仅当 $s = s_0$；
  - 对于任意程序状态 s, $(s_0, s) \in [\![c]\!]$ 当且仅当 $s = s_1$；

  因此，对于任意一个 F 的不动点 X 以及任意程序状态 s 都有：

  $(s_0, s) \in X$ 当且仅当 $(s_0, s) \in F(X)$
  当且仅当 $(s_0, s) \in \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ X \cup \text{test\_false}([\![e]\!])$
  当且仅当 $(s_1, s) \in X$
  当且仅当 $s = s_1$

  上面的最后一步用到了前面情形一的结论。

- 依此类推，不难发现，如果从程序状态 s_0 出发执行循环，并且在执行完 n 次循环体后到达 s_n 状态并离开循环，那么对于 F 的任意一个不动点 X 以及任意一个程序状态 s 都有：$(s_0, s) \in X$ 当且仅当 $s = s_n$。

- 不动点不唯一的例子
  - `while (true) do { skip }`
  - 循环体的语义：$[\![c]\!] = \text{id}$；
  - 循环条件恒为真：$\text{test\_true}([\![e]\!]) = \text{id}$；
  - 相应的：$\text{test\_false}([\![e]\!]) = \varnothing$；
  - 因此：
    $$F(X) \triangleq \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ X \cup \text{test\_false}([\![e]\!]) = X$$
  - 故，任意程序状态间的二元关系 X 都是 F 的不动点。

- 假设 s_0, s_1, s_2... 是一列无穷长的程序状态，满足：$[\![e]\!](s_i) = \mathbf{T}$ 并且 $(s_i, s_{i+1}) \in [\![c]\!]$，那么一个不动点 X 可能不包含任何一个形如 (s_i, s) 的程序状态有序对，也可能存在一个程序状态 s，使得某一个不动点 X 包含所有的 (s_0, s), (s_1, s), ... 这是因为：

  $(s_i, s) \in X$ 当且仅当 $(s_i, s) \in F(X)$，
  当且仅当 $(s_i, s) \in \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ X \cup \text{test\_false}([\![e]\!])$
  当且仅当 $(s_{i+1}, s) \in X$。

因此，不动点是不唯一的，而我们需要找的是在集合包含意义下，最小的不动点，也就是说：

- $[\![\text{while } (e) \text{ do } \{c\}]\!]$ 可以被定义为下述函数的最小不动点：

$$F(X) \triangleq \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ X \cup \text{test\_false}([\![e]\!])$$

9 Bourbaki-Witt 不动点定理

下面介绍 Bourbaki-Witt 不动点定理。这将统一的回答为什么有不动点，如何构造最小不动点。

- 定义：偏序集。满足下面三个条件的 $(A, \leqslant_A)$ 成为一个偏序集（partial ordering）：
  - 自反性：对于任意 $a \in A$， $a \leqslant_A a$
  - 传递性：对于任意 $a, b, c \in A$，如果 $a \leqslant_A b$、$b \leqslant_A c$，那么 $a \leqslant_A c$
  - 反对称性：对于任意 $a, b \in A$，如果 $a \leqslant_A b$、$b \leqslant_A a$，那么 $a = b$
- 例子：
  - $(\mathbb{R}, \leqslant)$ 是一个偏序集。
  - 如果 D 表示自然数之间的整除关系，即 $(a, b) \in D$ 当且仅当 $a \mid b$，那么 $(\mathbb{N}, D)$ 是一个偏序集。值得一提的是，在这个偏序关系下，两个自然数之间不一定可以相互比较；例如 $2 \nmid 3$ 并且 $3 \nmid 2$。
  - 如果 X 是一个集合，$\mathcal{P}(X)$ 表示 X 的幂集，那么 $(\mathcal{P}(X), \subseteq)$ 构成一个偏序集。
- 定义：完备偏序集。如果偏序集 $(A, \leqslant_A)$ 还满足下面性质，那么它是一个完备偏序集（complete partial ordering，CPO）：
  - 完备性：对于任意 $S \subseteq A$，如果 S 中任意两个元素之间都可以大小比较，那么 S 有上确界（least upper bound, lub），记做 $\mathrm{lub}(S)$，即：(1) 对于任意 $a \in S$，$a \leqslant_A \mathrm{lub}(S)$；(2) 如果某个 $b \in A$ 使得每一个 $a \in S$ 都有 $a \leqslant_A b$，那么 $\mathrm{lub}(S) \leqslant_A b$。

    上确界首先得是A中的元素！这点很重要，有时候可以推出它不是上确界。
  - 注：符合上述性质的 S 称为偏序集 A 上的一条链。
- 例子：
  - $(\mathbb{R}, \leqslant)$ 是偏序集但是不是完备偏序集，因为 $\mathbb{Z} \subseteq \mathbb{R}$ 是一条链，但是它没有上确界。
  - 如果 X 是一个集合，$\mathcal{P}(X)$ 表示 X 的幂集，那么 $(\mathcal{P}(X), \subseteq)$ 是一个完备偏序集。其中，对于任意 $U \subseteq \mathcal{P}(X)$，都有 $\mathrm{lub}(U) = \bigcup_{V \in U} V$ 是 U 的上确界。
  - 如果 D 表示自然数之间的整除关系，即 $(a, b) \in D$ 当且仅当 $a \mid b$，那么 $(\mathbb{N}, D)$ 是一个完备偏序集。特别的，如果 $U \subseteq \mathbb{N}$ 是一条链还是一个有穷集，那么 $\mathrm{lub}(U)$ 就是 U 的最小公倍数；如果 $U \subseteq \mathbb{N}$ 是一条链还是一个无穷集，那么 $\mathrm{lub}(U)$ 就是 0。
  - 如果 D^+ 表示正整数之间的整除关系，那么 $(\mathbb{Z}^+, D^+)$ 是一个偏序集，但不是一个完备偏序集。例如，$\{1, 2, 4, 8, ..., 2^n, ...\}$ 是整除关系上的一条链，但是它没有整除关系意义下的上确界。
- 定义：单调函数。如果 $(A, \leqslant_A)$ 是一个偏序集，那么 $F : A \to A$ 是一个单调函数当且仅当：对于任意 $a, b \in A$，如果 $a \leqslant_A b$，那么 $F(a) \leqslant_A F(b)$。
- 引理：如果 S 是偏序集 $(A, \leqslant_A)$ 上的一条链，$F : A \to A$ 是一个单调函数，那么 $F(S) \triangleq \{F(a) \mid a \in S\}$ 也是一条链。
- 证明：任给 $a, b \in S$，要么 $a \leqslant_A b$，要么 $b \leqslant_A a$。若前者成立，那么 $F(a) \leqslant_A F(b)$；若后者成立，那么 $F(b) \leqslant_A F(a)$，因此 $F(S)$ 中的元素间两两可以比较大小。
- 定义：单调连续函数。如果 $(A, \leqslant_A)$ 是一个完备偏序集，那么单调函数 $F : A \to A$ 是连续的当且仅当：对于任意一条非空链 S，$F(\mathrm{lub}(S)) = \mathrm{lub}(F(S))$。
- 引理：如果 $(A, \leqslant_A)$ 是一个完备偏序集，那么这个集合上有最小元，记做 $\bot$。

- 证明：空集 $\varnothing$ 是 $(A, \leqslant_A)$ 上的一条链。而任何一个 A 中元素，都是空集的上界。因此，对于任意 $a \in A$ 都有，$\text{lub}(\varnothing) \leqslant_A a$。

- 引理：如果 F 是完备偏序集 $(A, \leqslant_A)$ 上的单调连续函数，那么 $\{\bot, F(\bot), F(F(\bot)), ...\}$ 是 $(A, \leqslant_A)$ 上的一条链。

- 证明：

  由于 $\bot$ 是最小元，所以 $\bot \leqslant_A F(\bot)$。由于 F 是单调函数，所以 $F(\bot) \leqslant_A F(F(\bot))$。依次类推，$\bot \leqslant_A F(\bot) \leqslant_A F(F(\bot)) \leqslant_A ...$

- 定理：如果 F 是完备偏序集 $(A, \leqslant_A)$ 上的单调连续函数，$\text{lub}(\bot, F(\bot), F(F(\bot)), ...)$ 是 F 的一个不动点。

- 证明：

$$\begin{aligned} & F(\text{lub}(\bot, F(\bot), F(F(\bot)), ...)) \\ = \quad & \text{lub}(F(\bot), F(F(\bot)), F(F(F(\bot))), ...) \\ = \quad & \text{lub}(\bot, F(\bot), F(F(\bot)), F(F(F(\bot))), ...) \end{aligned}$$

- 定理：如果 F 是完备偏序集 $(A, \leqslant_A)$ 上的单调连续函数且 $F(a) = a$，那么 $\text{lub}(\bot, F(\bot), F(F(\bot)), ...) \leqslant_A a$。

- 证明：

$$\begin{aligned} & \bot \leqslant_A a \\ & F(\bot) \leqslant_A F(a) = a \\ & F(F(\bot)) \leqslant_A F(a) = a \\ & ... \end{aligned}$$

用 Bourbaki-Witt 不动点定义 while 语句语义

- $(\mathcal{P}(\text{state} \times \text{state}), \subseteq)$ 是一个完备偏序集；

- $F(X) \triangleq \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ X \cup \text{test\_false}([\![e]\!])$ 是一个单调连续函数；

  - $G(X) = Y \circ X$ 是单调连续函数；
  - $H(X) = X \cup Y$ 是单调连续函数；
  - 如果 $G(X)$ 与 $H(X)$ 都是单调连续函数，那么 $G(H(X))$ 也是单调连续函数；

- F 的最小不动点是：

$$\bigcup_{n \in \mathbb{N}} \left(F^{(n)}(\varnothing)\right)$$

两种定义的对应关系 $F^{(n)}(\varnothing) = \text{boundedLB}([\![e]\!], [\![c]\!])$。

10 Coq 中证明并应用 Bourbaki-Witt 不动点定理

下面我们将在 Coq 中证明 Bourbaki-Witt 不动点定理。在 Bourbaki-Witt 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.
```
A上的二元谓词。

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.
```
定义

```
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 Bourbaki-Witt 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'.
```
A’是X中的元素，那么。。。。

```
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.
```
A是下界

```
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 Bourbaki-Witt 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.
```

一个集合A，A上有一个序关系和等价关系

Class一般与结构的定义有关。

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。

偏序关系

```
Class PartialOrder_Setoid
        (A: Type) {RA: Order A} {EA: Equiv A}: Prop :=                Class的话是会去看注册列表中是否能推导出来一些参数
{
  PO_Reflexive_Setoid:> Reflexive_Setoid A;
  PO_Transitive:> Transitive order_rel; 提示Coq，有传递性
  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：

```
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。 *)
```

第一个参数做equiv变换

等价关系

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 Bourbaki-Witt 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).  n+1
```

```
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,          这个函数计算上确界
    increasing T -> is_omega_lub T (omega_lub T);
  bot_is_least: is_least bot  bot确实是最小元素
}.
```

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 Bourbaki-Witt 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。 *)
```

`BW_LFix` 定义了 Bourbaki-Witt 最小不动点。

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

先证明，`BW_LFix` 的计算结果确实是一个不动点。

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

再证明，`BW_LFix`的计算结果是最小不动点。

```
Lemma BW_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 ->
    BW_LFix f <= a.
(* 证明详见Coq源代码。 *)
```

接下去我们将利用 Bourbaki-Witt 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。

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

```
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。

```
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。

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

```
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。

```
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 Bourbaki-Witt 不动点定理的证明中用到了这一前提。

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

下面开始证明 `F(X) = (test1(D0) ∘ D ∘ while_denote D0 D) ∪ test0(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 Bourbaki-Witt 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 -> bool)
             (D: state -> state -> Prop):
  state -> state -> Prop :=
  BW_LFix (fun X => (test_true(D0) ∘ D ∘ X) ∪ test_false(D0)).
```

下面可以直接使用 Bourbaki-Witt 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。 *)
```

:>, 前提，Instance都是class会去找的参数