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。 *)
```

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。

```
Lemma while_sem_unroll1: forall D0 D1,
  while_sem D0 D1 ==
  test_true D0 ∘ D1 ∘ while_sem D0 D1 ∪ test_false D0.
Proof.
  intros.
  simpl.
  unfold while_sem.
  apply Sets_equiv_Sets_included; split.
  + apply Sets_indexed_union_included.
    intros.
    destruct n as [| n0]; simpl boundedLB.
    - apply Sets_empty_included.
    - rewrite <- (Sets_included_indexed_union _ _ n0).
      reflexivity.
  + apply Sets_union_included.
    - rewrite Rels_concat_indexed_union_distr_l.
      rewrite Rels_concat_indexed_union_distr_l.
      apply Sets_indexed_union_included; intros.
      rewrite <- (Sets_included_indexed_union _ _ (S n)).
      simpl.
      apply Sets_included_union1.
    - rewrite <- (Sets_included_indexed_union _ _ (S 0)).
      simpl boundedLB.
      apply Sets_included_union2.
Qed.
```

从这一结论可以看出，`while_sem D0 D1`是下面方程的一个解：

`X == test_true D0 ∘ D1 ∘ X ∪ test_false D0`;

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`while_sem D0 D1`是下面函数的一个不动点：