利用集合与关系定义指称语义

对于任意布尔表达式 e，我们规定它的语义 $[\![e]\!]$ 是程序状态集合的子集，是所有使得 e 求值为真的程序状态构成的集合。

- $[\![\mathrm{TRUE}]\!] = \mathrm{state}$
- $[\![\mathrm{FALSE}]\!] = \varnothing$
- $s \in [\![e_1 < e_2]\!]$ 当且仅当 $[\![e_1]\!](s) < [\![e_2]\!](s)$
- $[\![e_1 \&\& e_2]\!] = [\![e_1]\!] \cap [\![e_2]\!]$
- $[\![!e_1]\!] = \mathrm{state} \setminus [\![e_1]\!]$

在 Coq 中可以如下定义：

```
Definition true_sem: state -> Prop := Sets.full.
```

```
Definition false_sem: state -> Prop := ∅.
```

```
Definition lt_sem (D1 D2: state -> Z):
  state -> Prop :=
  fun s => D1 s < D2 s.
```

```
Definition and_sem (D1 D2: state -> Prop):
  state -> Prop :=
  D1 ∩ D2.
```

```
Definition not_sem (D: state -> Prop):
  state -> Prop :=
  Sets.complement D.
```

```
Fixpoint eval_expr_bool (e: expr_bool): state -> Prop :=
  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.
```

与整数类型表达式的行为等价定义一样，我们也可以用函数相等定义布尔表达式行为等价。

```
Definition bequiv (e1 e2: expr_bool): Prop :=
  ⟦ e1 ⟧ == ⟦ e2 ⟧.
```

下面先证明三个语义算子 `lt_sem` 、`and_sem` 与 `not_sem` 能保持函数相等，再利用函数相等的性质证明布尔表达式行为等价的性质。

```
#[export] Instance lt_sem_congr:
  Proper (func_equiv _ _ ==>
          func_equiv _ _ ==>
          Sets.equiv) lt_sem.
```

```
#[export] Instance and_sem_congr:
  Proper (Sets.equiv ==>
          Sets.equiv ==>
          Sets.equiv) and_sem.
```

```
#[export] Instance not_sem_congr:
  Proper (Sets.equiv ==> Sets.equiv) not_sem.
```

```
#[export] Instance bequiv_equiv: Equivalence bequiv.
```

```
#[export] Instance ELt_congr:
  Proper (iequiv ==> iequiv ==> bequiv) ELt.
```

```
#[export] Instance EAnd_congr:
  Proper (bequiv ==> bequiv ==> bequiv) EAnd.
```

```
#[export] Instance ENot_congr:
  Proper (bequiv ==> bequiv) ENot.
```

1 程序语句的指称语义定义

$(s_1, s_2) \in [\![c]\!]$ 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。

1.1 赋值语句

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

1.2 空语句

$$[\![\text{skip}]\!] = \{(s, s) \mid s \in \text{state}\}$$

1.3 顺序执行语句

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

1.4 条件分支语句

定义 1：

$$[\![\text{if } (e) \text{ then } \{c_1\} \text{ else } \{c_2\}]\!] = \begin{array}{l}(\{(s_1, s_2) \mid s_1 \in [\![e]\!]\} \cap [\![c_1]\!]) \cup \\ (\{(s_1, s_2) \mid s_1 \notin [\![e]\!]\} \cap [\![c_2]\!])\end{array}$$

定义 2：

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

其中，

$$\text{test\_true}([\![e]\!]) = \{(s_1, s_2) \mid s_1 \in [\![e]\!], s_1 = s_2\}$$
$$\text{test\_false}([\![e]\!]) = \{(s_1, s_2) \mid s_1 \notin [\![e]\!], s_1 = s_2\}。$$

1.5 循环语句语句

定义 1：

$$\text{iterLB}_0([\![e]\!], [\![c]\!]) = \text{test\_false}([\![e]\!]);$$
$$\text{iterLB}_{n+1}([\![e]\!], [\![c]\!]) = \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ \text{iterLB}_n([\![e]\!], [\![c]\!]);$$
$$[\![\text{while } (e) \text{ do } \{c\}]\!] = \bigcup_{n \in \mathbb{N}} \text{iterLB}_n([\![e]\!], [\![c]\!])。$$

定义 2：

$$\text{boundedLB}_0([\![e]\!], [\![c]\!]) = \varnothing$$
$$\text{boundedLB}_{n+1}([\![e]\!], [\![c]\!]) = \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ \text{boundedLB}_n([\![e]\!], [\![c]\!]) \cup \text{test\_false}([\![e]\!])$$
$$[\![\text{while } (e) \text{ do } \{c\}]\!] = \bigcup_{n \in \mathbb{N}} \text{boundedLB}_n([\![e]\!], [\![c]\!])$$

2 Coq 中定义程序语句的指称语义

```
Definition asgn_sem
             (X: var_name)
             (D: state -> Z)
             (s1 s2: state): Prop :=
  s2 X = D s1 /\ forall Y, X <> Y -> s2 Y = s1 Y.
```

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

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

```
Definition test_true
             (D: state -> Prop):
  state -> state -> Prop :=
  Rels.test D.
```

```
Definition test_false
             (D: state -> Prop):
  state -> state -> Prop :=
  Rels.test (Sets.complement D).
```

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

```
Fixpoint iterLB
           (D0: state -> Prop)
           (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.
```

```
(** 第一种定义方式 *)
Definition while_sem
             (D0: state -> Prop)
             (D1: state -> state -> Prop):
  state -> state -> Prop :=
  ⋃ (iterLB D0 D1).
```

```
Fixpoint boundedLB
           (D0: state -> Prop)
           (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.
```

```
(** 第二种定义方式 *)
Definition while_sem
             (D0: state -> Prop)
             (D1: state -> state -> Prop):
  state -> state -> Prop :=
  ⋃ (boundedLB D0 D1).
```

我们选择第二种定义。

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.
```

3 程序语句的行为等价

下面定义程序语句的行为等价。

```
Definition cequiv (c1 c2: com): Prop :=
  ⟦ c1 ⟧ == ⟦ c2 ⟧.
```

可以证明，赋值语句、顺序执行、if 语句和 while 语句对应的语义算子 `asgn_sem` 、`seq_sem` 、`if_sem` 与 `while_sem` 能由相同的函数及集合计算得到相同的集合。其中，证明 if 语句和 while 语句性质时，需要先证明 `test_true` 和 `test_false` 能够由相同的函数计算得到相同的集合。

```
#[export] Instance asgn_sem_congr:
  Proper (eq ==> func_equiv _ _ ==> Sets.equiv) asgn_sem.
```

```
#[export] Instance seq_sem_congr:
  Proper (Sets.equiv ==> Sets.equiv ==> Sets.equiv) seq_sem.
```

```
#[export] Instance test_true_congr:
  Proper (Sets.equiv ==> Sets.equiv) test_true.
```

```
#[export] Instance test_false_congr:
  Proper (Sets.equiv ==> Sets.equiv) test_false.
```

```
#[export] Instance if_sem_congr:
  Proper (Sets.equiv ==>
          Sets.equiv ==>
          Sets.equiv ==>
          Sets.equiv) if_sem.
```

下面证明 Simplewhile 程序语句行为等价的代数性质。

```
#[export] Instance cequiv_equiv: Equivalence cequiv.
```

```
#[export] Instance CAsgn_congr:
  Proper (eq ==> iequiv ==> cequiv) CAsgn.
```

```
#[export] Instance CSeq_congr:
  Proper (cequiv ==> cequiv ==> cequiv) CSeq.
```

```
#[export] Instance CIf_congr:
  Proper (bequiv ==> cequiv ==> cequiv ==> cequiv) CIf.
```

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, [[ c; skip ]] ~=~ c.
```

```
Lemma skip_seq:
  forall c, [[ skip; c ]] ~=~ c.
```

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,
  [[ (c1; c2); c3 ]] ~=~ [[ c1; (c2; c3) ]].
```

4 复杂程序语句语义性质的证明

下面证明几条程序语句语义的一般性性质。我们首先可以证明，两种 while 语句语义的定义方式是等价的。

```
Lemma while_sem1_while_sem2_equiv:
  forall D0 D1,
    WhileSem1.while_sem D0 D1 ==
    WhileSem2.while_sem D0 D1.
```

还可以证明，`boundedLB` 是递增的。

```
Theorem boundedLB_inc: forall D0 D1 n m,
  boundedLB D0 D1 m ⊆ boundedLB D0 D1 (n + m).
```

```
#[export] Instance while_sem_congr:
  Proper (Sets.equiv ==>
          Sets.equiv ==>
          Sets.equiv) while_sem.
```

```
#[export] Instance CWhile_congr:
  Proper (bequiv ==> cequiv ==> cequiv) CWhile.
```

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_unroll1: forall e c,
  [[ while (e) do {c} ]] ~=~
  [[ if (e) then { c; while (e) do {c} } else {skip} ]].
```

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

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