基于关系的指称语义

1 用二元关系表示程序语句的语义

$(s_1, s_2) \in [\![c]\!]$ 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。

2 二元关系的运算

$(x, z) \in R_1 \circ R_2$ 当且仅当存在 y 使得 $(x, y) \in R_1$ 并且 $(y, z) \in R_2$。
$I_A = \{(a, a) \mid a \in A\}$。
如果 $X \subseteq A$，那么 $\mathrm{test}(X) = \{(a, a) \mid a \in X\}$。

3 在 Coq 中表示集合与二元关系

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 这一拓展库中也直接写作 `a ∈ X` 。

类似的，在 Coq 中也常常使用 `R: A -> B -> Prop` 来表示 `A` 与 `B` 中元素之间的二元关系。数学上，“`(a, b)` 是集合 `R` 中的元素”可以写作 `(a,b) ∈ R` ，SetsClass 拓展库也支持我们这么表达，当然，它在 Coq 中的实际定义就是 `R a b` 这个命题成立。

SetsClass 拓展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` ；
- 全集：定义为 `Sets.full` （全集没有专门的表示符号）；
- 单元集：用一对方括号表示，定义为 `Sets.singleton` ；
- 补集：定义为 `Sets.complement` （补集没有专门的表示符号）；
- 并集：用 `∪` 表示，定义为 `Sets.union` ；
- 交集：用 `∩` 表示，定义为 `Sets.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` ；

在这些符号中，补集以及其他 Coq 函数的优先级最高，交集的优先级其次，并集的优先级再次，集合相等、集合包含与属于号的优先级最低。

SetsClass 拓展库中提供了这些关于二元关系的定义：

- 二元关系的连接：用 `∘` 表示，定义为 `Rels.concat` ；
- 相等关系：定义为 `Rels.id` （没有专门的表示符号）；
- 测试：定义为 `Rels.test` （没有专门的表示符号）。

4 程序语句的指称语义定义

4.1 赋值语句

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

4.2 空语句

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

4.3 顺序执行语句

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

4.4 条件分支语句

定义 1：

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

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，

$$\begin{aligned}&\text{test\_true}([\![e]\!]) = \{(s_1, s_2) \mid [\![e]\!](s_1) = \mathbf{T}, s_1 = s_2\} \\ &\text{test\_false}([\![e]\!]) = \{(s_1, s_2) \mid [\![e]\!](s_1) = \mathbf{F}, s_1 = s_2\}。\end{aligned}$$

4.5 循环语句语句

定义 1：

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

定义 2：

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

5 Coq 中定义程序语句的指称语义

```
Record asgn_sem
         (X: var_name)
         (D: state -> Z)
         (s1 s2: state): Prop :=
  {
    asgn_sem_asgn_var: s2 X = D s1;
    asgn_sem_other_var: 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 -> bool):
  state -> state -> Prop :=
  Rels.test (fun s => D s = true).
```

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

```
(** 第一种定义方式 *)
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.
```

```
(** 第二种定义方式 *)
Definition while_sem
             (D0: state -> bool)
             (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.
```

6 集合、关系与逻辑命题的 Coq 证明

6.1 集合命题的基本证明方法

`Sets_unfold` 指令可以将集合的性质转化为逻辑命题。

集合相等是一个等价关系，集合包含具有自反性和传递性。集合间的交集、并集和补集运算保持“包含”与“被包含”关系，也会保持集合相等关系。SetsClass 拓展库提供了这些性质的证明，从而支持利用 `rewrite` 指令证明集合性质。

6.2 交集与并集性质的 Coq 证明

证明集合相等的方法：

```
Sets_equiv_Sets_included:
  forall x y, x == y <-> x ⊆ y /\ y ⊆ x
```

证明交集有关的性质：

- `x ⊆ y ∩ z` 可以被规约为 `x ⊆ y` 与 `x ⊆ z`；
- `x ∩ y ⊆ z` 可以被规约为 `x ⊆ z`；
- `x ∩ y ⊆ z` 也可以被规约为 `y ⊆ z`。

在 Coq 中，前一种证明可以通过 `apply` 下面引理实现。

```
Sets_included_intersect:
  forall x y z, x ⊆ y -> x ⊆ z -> x ⊆ y ∩ z
```

而后两种证明可以通过 `rewrite` 下面引理实现。

```
Sets_intersect_included1:
  forall x y, x ∩ y ⊆ x
Sets_intersect_included2:
  forall x y, x ∩ y ⊆ y
```

```
  (** ... 证明详见Coq源代码 ... *)
  (** ... 证明详见Coq源代码 ... *)
  (** ... 证明详见Coq源代码 ... *)
```

证明并集有关的性质：

`x ⊆ y ∪ z` 可以被规约为 `x ⊆ y`；
`x ⊆ y ∪ z` 也可以被规约为 `x ⊆ z`；
`x ∪ y ⊆ z` 可以被规约为 `x ⊆ z` 与 `y ⊆ z`。

在 Coq 中，前两种证明可以通过从右向左 `rewrite` 下面引理实现。

```
Sets_included_union1:
  forall x y, x ⊆ x ∪ y
Sets_included_union2:
  forall x y, y ⊆ x ∪ y
```

而后一种证明则可以通过 `apply` 下面引理实现。

```
Sets_union_included:
  forall x y z, x ⊆ z -> y ⊆ z -> x ∪ y ⊆ z;
```

有时，包含号 `⊆` 左侧的集合不是一个并集，需要先使用交集对于并集的分配律才能使用 `Sets_union_included`。

基本证明方法汇总：

```
Sets_equiv_Sets_included:
  forall x y, x == y <-> x ⊆ y /\ y ⊆ x
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:
  forall x y z, x ⊆ z -> y ⊆ z -> x ∪ y ⊆ z
Sets_intersect_union_distr_r:
  forall x y z, (x ∪ y) ∩ z == x ∩ z ∪ y ∩ z
Sets_intersect_union_distr_l:
  forall x y z, x ∩ (y ∪ z) == x ∩ y ∪ x ∩ z
```

其他常用性质汇总：

```
Sets_intersect_comm:
  forall x y, x ∩ y == y ∩ x
Sets_intersect_assoc:
  forall x y z, (x ∩ y) ∩ z == x ∩ (y ∩ z)
Sets_union_comm:
  forall x y, x ∪ y == y ∪ x
Sets_union_assoc:
  forall x y z, (x ∪ y) ∪ z == x ∪ (y ∪ z)
Sets_union_intersect_distr_l:
  forall x y z, x ∪ (y ∩ z) == (x ∪ y) ∩ (x ∪ z)
Sets_union_intersect_distr_r:
  forall x y z, (x ∩ y) ∪ z == (x ∪ z) ∩ (y ∪ z)
```

6.3 空集、全集、无穷交与无穷并性质的 Coq 证明

SetsClass 拓展库对于空集的支持主要是通过以下性质：空集是一切集合的子集。

```
Sets_empty_included: forall x, ∅ ⊆ x
```

相对应的，一切集合都是全集的子集。

```
Sets_included_full: forall x, x ⊆ Sets.full
```

基于这两条性质，可以证明许多有用的导出性质。SetsClass 提供的导出性质有：

```
Sets_union_empty_l: forall x, ∅ ∪ x == x
Sets_union_empty_r: forall x, x ∪ ∅ == x
Sets_intersect_empty_l: forall x, ∅ ∩ x == ∅
Sets_intersect_empty_r: forall x, x ∩ ∅ == ∅
Sets_union_full_l: forall x, Sets.full ∪ x == Sets.full
Sets_union_full_r: forall x, Sets.full ∪ ∅ == Sets.full
Sets_intersect_full_l: forall x, Sets.full ∩ x == x
Sets_intersect_full_r: forall x, x ∩ Sets.full == x
Sets_equiv_empty_fact: forall x, x ⊆ ∅ <-> x == ∅
Sets_equiv_full_fact: forall x, Sets.full ⊆ x <-> x == Sets.full
```

SetsClass 拓展库提供了两种支持无穷交集和无穷并集的定义。它们的证明方式与普通的并集与交集的证明方式是类似的。

- 基于指标集的集合并：`⋃ X`，`Sets.indexed_union X`

$$\{x \mid \exists i \in I.\ X_i\}$$

- 基于指标集的集合交：`⋂ X`，`Sets.indexed_intersect X`

$$\{x \mid \forall i \in I.\ X_i\}$$

- 广义并：`⨆ U`，`Sets.general_union U`

$$\{x \mid \exists X \in U.\ x \in X\}$$

- 广义交：`⨅ U`，`Sets.general_intersect U`

$$\{x \mid \forall X \in U.\ x \in X\}$$

下面是 SetsClass 库中已经证明的关于无穷交集与无穷并集（基于指标集）的性质。

```
Sets_included_indexed_intersect:
  forall xs y, (forall n, y ⊆ xs n) -> y ⊆ ⋂ xs
Sets_included_indexed_union:
  forall n xs x, xs n ⊆ ⋃ xs
Sets_indexed_union_included:
  forall xs y, (forall n, xs n ⊆ y) -> ⋃ xs ⊆ y
Sets_indexed_intersect_included:
  forall n xs, ⋂ xs ⊆ xs n
Sets_intersect_indexed_union_distr_r:
 forall xs y, ⋃ xs ∩ y == ⋃ (fun n => xs n ∩ y)
Sets_intersect_indexed_union_distr_l:
 forall x ys, x ∩ ⋃ ys == ⋃ (fun n => x ∩ ys n)
```

下面是广义交与广义并的性质。

```
Sets_general_intersect_included:
  forall U x, U x -> ⋂ U ⊆ x
Sets_general_union_included:
  forall U y, (forall x, U x -> x ⊆ y) -> ⋃ U ⊆ y
Sets_included_general_union:
  forall U x, U x -> x ⊆ ⋃ U
Sets_included_general_intersect:
  forall U y, (forall x, U x -> y ⊆ x) -> y ⊆ ⋂ U
```

6.4 逻辑命题的 Coq 证明

下面是证明“并且”、“或”与“当且仅当”时常用的证明指令汇总与拓展。

Coq 证明脚本 1. left 指令与 right 指令. 如果待证明结论具有 `P \/ Q` 的形式，那么 `left` 可以将该结论规约为 `P` ，`right` 可以将该结论规约为 `Q` 。

Coq 证明脚本 2. 命题逻辑证明中的 split 指令. 如果待证明结论具有 `P /\ Q` 的形式，那么 `split` 可以将当前证明目标规约为两个更简单的证明目标，它们的前提与原证明目标的前提相同，它们的结论分别为 `P` 与 `Q` 。

Coq 证明脚本 3. 命题逻辑证明中的 destruct 指令. 如果证明时前提 `H` 具有形式 `P /\ Q` 或具有形式 `P \/ Q` ，则可以使用 `destruct H` 指令。当 `H` 具有形式 `P /\ Q` 时，该指令会将此前提分解为两个前提 `P` 与 `Q` 。当 `H` 具有形式 `P \/ Q` 时，该指令会将当前证明目标规约为两个证明目标，其中一个将前提 `H` 被改为 `P` ，另一个将前提 `H` 改为 `Q` 。

Coq 允许用户使用 `destruct ... as ...` 指令对 destruct 得到的新前提手动重命名。例如当 `H` 具有形式 `P /\ Q` 时，`destruct H as [H1 H2]` 指令将生成下面两个证明前提：

```
H1: P
H2: Q
```

又例如当 `H` 具有形式 `P \/ Q` 时，`destruct H as [H1 | H2]` 指令也可以用于手动命名。与 intros 指令中的规定一样，当需要对 destruct 结果中的一部分手动命名而对另一部分自动命名时，可以使用问号 `?` 表示那些需要由 Coq 自动命名的名字。

Coq 允许用户对多个前提同时执行 destruct 指令,例如 `destruct H, H0` 就表示先 `destruct H` 再 `destruct H0` 。Coq 也允许用户在一条 destruct 指令中对 destruct 的结果再进一步分解或进一步分类讨论，但具体需要分解多少层，需要使用 `destruct ... as ...` 指令做具体说明。例如，当 `H` 具有形式

```
(P /\ Q) /\ (R /\ S)
```

时，`destruct H as [? [? ?]]` 指令会将 `H` 分解为 `P /\ Q` 、`R` 与 `S` ；而 `destruct H as [[? ?] ?]` 指令会将 `H` 分解为 `P` 、`Q` 与 `R /\ S` 。另外，对“并且”与“或”的分解与分类讨论也可以相互嵌套，例如，当 `H` 具有形式

```
(P /\ Q) \/ R
```

时，可以使用 `destruct H as [[HP HQ] | HR]` 指令在分类讨论的同时对其中一个分类讨论的分支对前提做进一步分解。

Coq 证明脚本 4. 引入模式. 前面已经介绍，`destruct ... as ...` 指令可以一次性完成若干次的命题拆解或分类讨论。在这一指令中，`as` 之后的结构成为“引入模式”（intro-pattern）。以下是与目前所学相关的几种引入模式：

- 针对“并且”的命题拆解 `[intro_pattern1 intro_pattern2]` ；
- 针对“或”的分类讨论 `[intro_pattern1 | intro_pattern2]` ；
- 新引入的名字，例如 `H` ；
- 表示由 Coq 系统自动命名的问号 `?` ；
- 表示直接丢弃拆分结果的下划线 `_` ；

除了 destruct 指令之外，intros 指令与 pose proof 指令也可以使用引入模式，在完成原有功能的基础上，再进行一次 destruct。例如，`intros [H1 H2] [H3 | H3]` 相当于依次执行：

```
intros H1 H2
destruct H1 as [H1 H2]
destruct H3 as [H3 | H3] ;
```

而 `pose proof H x y as [H1 H2]` 相当于依次执行：

```
pose proof H x y as H1
destruct H1 as [H1 H2] 。
```

Coq 证明脚本 5. 关于“存在”的证明. 当待证明结论形为：“存在一个 `x` 使得...”，那么可以用 `exists` 指明究竟哪个 `x` 使得该性质成立。

当某前提形为：“存在一个 `x` 使得...”，那么可以使用 Coq 中的 `destruct` 指令进行证明。这一证明指令相当于数学证明中的：任意给定一个这样的 `x` 。

6.5 二元关系运算性质的 Coq 证明

二元关系的运算性质：- 结合律：`(x ∘ y) ∘ z == x ∘ (y ∘ z)`
- 左单位元：`Rels.id ∘ x == x`
- 右单位元：`x ∘ Rels.id == x`
- 左分配律：`x ∘ (y ∪ z) == x ∘ y ∪ x ∘ z`
- 右分配律：`(x ∪ y) ∘ z == x ∘ z ∪ y ∘ z`

另外，二元关系对并集的分配律对于无穷并集也成立。

```
  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
  Rels_concat_union_distr_l:
    forall x y1 y2, x ∘ (y1 ∪ y2) == x ∘ y1 ∪ x ∘ y2
  Rels_concat_union_distr_r:
    forall x1 x2 y, (x1 ∪ x2) ∘ y == x1 ∘ y ∪ x2 ∘ y
  Rels_concat_indexed_union_distr_l:
    forall x ys, x ∘ ⋃ ys == ⋃ (fun n => x ∘ ys n)
  Rels_concat_indexed_union_distr_r:
    forall xs y, ⋃ xs ∘ y == ⋃ (fun n => xs n ∘ y)
```

测试关系 `Rels.test` 有下面几条重要性质。

```
  Rels_test_full: Rels.test Sets.full == Rels.id
  Rels_test_empty: Rels.test ∅ == ∅
  Rels_test_is_id:
    forall x, x == Sets.full -> Rels.test x == Rels.id
  Rels_test_is_empty:
    forall x, x == ∅ -> Rels.test x == ∅
```

7 程序语句指称语义的性质

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).
```

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 (func_equiv _ _ ==> Sets.equiv) test_true.
```

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

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

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

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

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) ]].
```

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：

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

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,
  remove_skip c ~=~ c.
```