指称语义

1 简单表达式的指称语义

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，我们可以如下定义程序状态集合：

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

进一步，整数类型表达式 e 的行为可以被定义为 e 在每个程序状态上的值。

$[\![e]\!] : \text{state} \to \mathbb{Z}$ 是一个程序状态到整数的函数；
$[\![e]\!](s)$ 表示表达式 e 在程序状态 s 上的求值结果。

基于这一设定，可以写出下面的具体定义：

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

其中 $s \in \text{state}$。

上面这些式子可以写成下面这些 Coq 代码。

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

```
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.
```

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。

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

这一定义写到 Coq 中便是下面这个整数类型表达式之间的二元关系。

```
Definition iequiv (e1 e2: expr_int): Prop :=
  forall s, ⟦ e1 ⟧ s = ⟦ e2 ⟧ s.
```

之后我们将在 Coq 中用 `e1 ~=~ e2` 表示 `iequiv e1 e2` 。

习题 1. 请证明下面 SimpleWhile 中整数类型表达式的行为等价。

```
Lemma plus_plus_assoc:
  forall a b c: expr_int,
    [[ a + (b + c) ]] ~=~ [[ a + b + c ]].
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Lemma plus_minus_assoc:
  forall a b c: expr_int,
    [[ a + (b - c) ]] ~=~ [[ a + b - c ]].
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Lemma minus_plus_assoc:
  forall a b c: expr_int,
    [[ a - (b + c) ]] ~=~ [[ a - b - c ]].
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Lemma minus_minus_assoc:
  forall a b c: expr_int,
    [[ a - (b - c) ]] ~=~ [[ a - b + c ]].
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

3 Coq 代数结构：等价关系

先前我们利用 Coq 归纳类型与递归函数定义了二叉树 `tree` 与二叉树结构相等 `same_structure` 。我们还证明过， `same_structure` 具有传递性（ `same_structure_trans` ），事实上，我们还知道 `same_structure` 是一个等价关系！数学上，一个二元关系“≡”是一个等价关系当且仅当它满足下面三个性质：

- 自反性： `forall a, a ≡ a`
- 对称性： `forall a b, a ≡ b -> b ≡ a`
- 传递性： `forall a b c, a ≡ b -> b ≡ c -> a ≡ c`

Coq 标准库提供了自反、对称、传递与等价的统一定义，并基于这些统一定义提供了 `rewrite` 、 `reflexivity` 等证明指令支持。下面三条证明中， `Reflexive` 、 `Symmetric` 与 `Transitive` 是 Coq 标准库对于自反、对称与传递的定义。Coq 标准库还将这三个定义注册成了 Coq 的 Class，这使得 Coq 能够提供一些特定的证明支持。这里的关键字也不再使用 `Lemma` 或 `Theorem` ，而是使用 `Instance` ，这表示 Coq 将在后续证明过程中为 `same_structure` 提供自反、对称与传递相关的专门支持。

Coq 还将这三条性质打包起来，定义了等价关系 `Equivalence` 。要在 Coq 中证明 `same_structure` 是一个等价关系，可以使用 `split` 指令，将“等价关系”规约为“自反性”、“对称性”与“传递性”。

4 Coq 代数结构：Morphism

Coq 标准库提供了 `Proper` 表示“保持等价性”。

```
Definition any {A: Type} (a b: A): Prop := True.
```

```
#[export] Instance Node_same_structure_morphism:
  Proper (same_structure ==>
          any ==>
          same_structure ==>
          same_structure) Node.
```

这个性质说得是：`Node` 是一个三元函数，如果对其第一个参数做 `same_structure` 变换，对其第二个参数做任意变换，同时对其第三个参数做 `same_structure` 变换，那么这个三元函数的计算结果也会做 `same_structure` 变换。在证明这一结论时，需要展开 `Proper` 的定义，还需要展开 `==>` 的定义，它的 Coq 名字是 `respectful` 。

```
Proof.
  intros.
  unfold Proper, respectful.
  intros t11 t21 ? n1 n2 _ t12 t22 ?.
  simpl.
  tauto.
Qed.
```

下面补充证明， `any` 是一个自反关系。

```
#[export] Instance any_refl: forall A: Type, Reflexive (@any A).
(* 证明详见Coq源代码。 *)
```

5 行为等价的性质

整数类型表达式之间的行为等价符合下面几条重要的代数性质。

```
#[export] Instance iequiv_refl: Reflexive iequiv.
```

```
#[export] Instance iequiv_symm: Symmetric iequiv.
```

```
#[export] Instance iequiv_trans: Transitive iequiv.
```

```
#[export] Instance iequiv_equiv: Equivalence iequiv.
```

```
#[export] Instance EAdd_iequiv_morphism:
  Proper (iequiv ==> iequiv ==> iequiv) EAdd.
```

```
#[export] Instance ESub_iequiv_morphism:
  Proper (iequiv ==> iequiv ==> iequiv) ESub.
```

```
#[export] Instance EMul_iequiv_morphism:
  Proper (iequiv ==> iequiv ==> iequiv) EMul.
```

6 函数与等价关系

对于类型 `B` 上的二元关系 `R`，可以定义二元关系 `pointwise_relation A R`：

```
Definition pointwise_relation
             (A: Type) {B: Type}
             (R: B -> B -> Prop)
             (f g: A -> B): Prop :=
  forall a: A, R (f a) (g a).
```

Coq 标准库也证明了，如果 `R` 是等价关系，那么 `pointwise_relation A R` 也是等价关系。

```
#[export] Instance pointwise_equivalence:
  forall {A B: Type} {R: B -> B -> Prop},
    Equivalence R ->
    Equivalence (pointwise_relation A R).
(* 证明详见Coq源代码。 *)
```

在 Coq 中，普通的等号 `=` 实际是一个 Notation，其背后的定义名称为 `eq`。这是一个多态二元谓词，例如 `@eq Z` 表示“整数相等”，`@eq (list Z)` 表示“整数列表相等”。这个等号表示的“相等”自然也是一个等价关系，这一定理在 Coq 标准库中的描述如下：

```
eq_equivalence: forall {A : Type}, Equivalence (@eq A)
```

更进一步，两个类型为 `A -> B` 的函数，“它们在 `A` 类型的自变量任意取值时求值结果都相同”就可以用下面二元关系表示：

```
Definition func_equiv (A B: Type):
  (A -> B) -> (A -> B) -> Prop :=
  pointwise_relation A (@eq B).
```

我们知道，`func_equiv` 也一定是一个等价关系。

```
#[export] Instance func_equiv_equiv:
  forall A B, Equivalence (func_equiv A B).
Proof.
  intros.
  apply pointwise_equivalence.
  apply eq_equivalence.
Qed.
```

除了可以定义函数之间的等价关系之外，还可以反过来利用函数构造等价关系。

```
Theorem equiv_in_domain:
  forall {A B: Type} (f: A -> B) (R: B -> B -> Prop),
    Equivalence R ->
    Equivalence (fun a1 a2 => R (f a1) (f a2)).
(* 证明详见Coq源代码。 *)
```

7 利用高阶函数定义指称语义

先定义三个算术运算符对应的语义算子：

```
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.
```

这意味着整数类型表达式的语义满足下面性质：

- $[\![e_1 + e_2]\!] = \text{add\_sem}([\![e_1]\!], [\![e_2]\!])$
- $[\![e_1 - e_2]\!] = \text{sub\_sem}([\![e_1]\!], [\![e_2]\!])$
- $[\![e_1 * e_2]\!] = \text{mul\_sem}([\![e_1]\!], [\![e_2]\!])$

可以证明，这三个语义函数都能保持函数相等。

```
#[export] Instance add_sem_congr:
  Proper (func_equiv _ _ ==>
          func_equiv _ _ ==>
          func_equiv _ _) add_sem.
```

```
#[export] Instance sub_sem_congr:
  Proper (func_equiv _ _ ==>
          func_equiv _ _ ==>
          func_equiv _ _) sub_sem.
```

```
#[export] Instance mul_sem_congr:
  Proper (func_equiv _ _ ==>
          func_equiv _ _ ==>
          func_equiv _ _) mul_sem.
```

基于上面这三个用高阶函数定义的语义算子，可以重新定义整数类型表达式的指称语义。

```
Definition const_sem (n: Z): state -> Z :=
  fun s => n.
```

```
Definition var_sem (X: var_name): state -> Z :=
  fun s => 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.
```

同时，我们也可以用函数相等来定义表达式行为等价和并利用函数相等的代数性质来证明行为等价的代数性质。

```
Definition iequiv (e1 e2: expr_int): Prop :=
  (⟦ e1 ⟧ == ⟦ e2 ⟧)%func.
```

```
#[export] Instance iequiv_equiv: Equivalence iequiv.
```

```
#[export] Instance EAdd_congr:
  Proper (iequiv ==> iequiv ==> iequiv) EAdd.
```

```
#[export] Instance ESub_congr:
  Proper (iequiv ==> iequiv ==> iequiv) ESub.
```

```
#[export] Instance EMul_congr:
  Proper (iequiv ==> iequiv ==> iequiv) EMul.
```

8 布尔表达式的语义

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: state -> bool :=
  fun s => true.
```

```
Definition false_sem: state -> bool :=
  fun s => false.
```

```
Definition lt_sem (D1 D2: state -> Z):
  state -> bool :=
  fun s =>
    if Z_lt_dec (D1 s) (D2 s)
    then true
    else false.
```

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

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

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 ⟧)%func.
```

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 _ _ ==>
          func_equiv _ _) lt_sem.
```

```
#[export] Instance and_sem_congr:
  Proper (func_equiv _ _ ==>
          func_equiv _ _ ==>
          func_equiv _ _) and_sem.
```

```
#[export] Instance not_sem_congr:
  Proper (func_equiv _ _ ==> func_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.
```