表达式指称语义

1 SimpleWhile 整数类型表达式的指称语义

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 定义有符号 64 位运算的表达式语义

在表达式的指称语义中表示表达式求值错误（有符号 64 位整数的运算越界的情况）这一概念，数学上有两种常见方案。其一是将求值结果由“整数”改为“整数或求值失败”。

- 原指称语义：

$$\forall e.\quad [\![e]\!] : \text{state} \to \mathbb{Z}$$

- 新指称语义：

$$\forall e.\quad [\![e]\!] : \text{state} \to \mathbb{Z} \cup \{\epsilon\}$$

e:undefined

- 程序状态：

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

$$s \in \text{state} \text{ 合法当且仅当 } \forall x.\ -2^{63} \leqslant s(x) \leqslant 2^{63} - 1$$

- $[\![n]\!](s) = n$，若 $-2^{63} \leqslant n \leqslant 2^{63} - 1$
- $[\![n]\!](s) = \epsilon$，若 $-2^{63} \leqslant n \leqslant 2^{63} - 1$ 不成立
- $[\![x]\!](s) = s(x)$
- 若 $[\![e_1]\!](s) \neq \epsilon$，$[\![e_2]\!](s) \neq \epsilon$ 并且 $-2^{63} \leqslant [\![e_1]\!](s) + [\![e_2]\!](s) \leqslant 2^{63} - 1$
  $[\![e_1 + e_2]\!](s) = [\![e_1]\!](s) + [\![e_2]\!](s)$
- 若否
  $[\![e_1 + e_2]\!](s) = \epsilon$
- $[\![e_1 - e_2]\!](s) = ...$
- $[\![e_1 * e_2]\!](s) = ...$

3 变量初始化与表达式语义

- $s \in \text{state}$ 当且仅当 s 是这样的一个二元组 $s = (s_{\text{status}}, s_{\text{value}})$，其中

$$s_{\text{status}} : \text{var\_name} \to \{\text{I}, \text{U}\},$$
$$s_{\text{value}} : \text{var\_name} \to \mathbb{Z} \cup \{\epsilon\}$$

- $s \in \text{state}$ 是一个合法状态当且仅当

$$\forall x.\ s_{\text{status}}(x) = \text{I} \Rightarrow -2^{63} \leqslant s_{\text{value}}(x) \leqslant 2^{63} - 1$$
$$\forall x.\ s_{\text{status}}(x) = \text{U} \Rightarrow s_{\text{value}}(x) = \text{U}$$

- $[\![e]\!] : \text{state} \to \mathbb{Z} \cup \{\epsilon\}$

4 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。 *)
```

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 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：

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

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.
```

可以证明，前面定义的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.
```

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.
```