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：

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

下面使用 Coq 递归函数定义它的行为。

```
Fixpoint eval_expr_int (e: expr_int) (st: state) : Z :=
  match e with
  | EConst n => n
  | EVar X => st X
  | EAdd e1 e2 => eval_expr_int e1 st + eval_expr_int e2 st
  | ESub e1 e2 => eval_expr_int e1 st - eval_expr_int e2 st
  | EMul e1 e2 => eval_expr_int e1 st * eval_expr_int e2 st
  end.
```

下面是两个具体的例子。

```
Example eval_example1: forall (st: state),
  st "x" = 1 ->
  st "y" = 2 ->
  eval_expr_int [["x" + "y"]] st = 3.
Proof. intros. simpl. rewrite H, H0. reflexivity. Qed.
```

```
Example eval_example2: forall (st: state),
  st "x" = 1 ->
  st "y" = 2 ->
  eval_expr_int [["x" * "y" + 1]] st = 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。

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

```
Definition add_sem (D1 D2: state -> Z) (st: state): Z :=
  D1 st + D2 st.
```

```
Definition sub_sem (D1 D2: state -> Z) (st: state): Z :=
  D1 st - D2 st.
```

```
Definition mul_sem (D1 D2: state -> Z) (st: state): Z :=
  D1 st * D2 st.
```

```
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) (st: state): Z := n.
Definition var_sem (X: var_name) (st: state): Z := st 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 更多高阶函数的例子

```
Definition doit3times {X:Type} (f:X->X) (n:X) : X :=
  f (f (f n)).
```

这里，`f`这个参数本身也是一个函数（从`X`到`X`的函数）而`doit3times`则把`f`在`n`上作用了 3 次。

```
Definition minustwo (x: Z): Z := x - 2.
```

- **习题 1.** `doit3times minustwo 9`的值是多少?
- **习题 2.** `doit3times minustwo (doit3times minustwo 9)`的值是多少?
- **习题 3.** `doit3times (doit3times minustwo) 9`的值是多少?
- **习题 4.** `doit3times doit3times minustwo 9`的值是多少?

Coq 中允许用户使用`fun`关键字表示匿名函数，例如:

```
Example fact_doit3times_anon1:
  doit3times (fun x => x - 2) 9 = 3.
Proof. reflexivity. Qed.
```

```
Example fact_doit3times_anon2:
  doit3times (fun x => x * x) 2 = 256.
Proof. reflexivity. Qed.
```

这里`fun x => x - 2`与之前定义的`minustwo`是相同的，而`fun x => x * x`则表示了平方这样一个函数。

- **习题 5.** `doit3times (fun x => - x) 5`的值是多少?

习题 6. `doit3times (fun f x y => f y x) (fun x y => x - y) 1 2`的值是多少?

- **习题 7.** 如果`Func_add`定义为`fun f g x => f x + g x`那么 `doit3times (Func_add minustwo) (fun x => x * x)`相当于什么函数?

- **习题 8.** `doit3times ((fun x y => y * y - x * y + x * x) 1) 1`的值是多少?