现实程序语言的指称语义

1 表示 64 位整数运算的整数表达式语义

程序状态的修改：

$$\text{原程序状态：} \quad \text{state} \triangleq \text{var\_name} \to \mathbb{Z}$$
$$\text{新程序状态：} \quad \text{state} \triangleq \text{var\_name} \to \mathbb{Z}_{2^{64}}$$

```
Definition state: Type := var_name -> int64.
```

这里`int64`是 CompCert 库中定义的 64 位整数，该定义是`compcert.lib.Integers`这一头文件导入的。除了`int64`的类型定义，CompCert 还定义了 64 位整数的运算并证明相关运算的一些基本性质，例如`Int64.add`表示 64 位算术加法，`Int64.and`表示按位做『与』运算。

除了上述表达算术运算、位运算的函数外，还有 3 个 64 位整数相关的函数十分常用，分别是：`Int64.repr`，`Int64.signed`，`Int64.unsigned`。另外，下面几个常数定义了有符号 64 位整数与无符号 64 位整数的大小边界：`Int64.max_unsigned`，`Int64.max_signed`，`Int64.min_signed`。

除了修改程序状态的定义，还需要相应修改程序证整数类型表达式的语义。

$$\text{原指称语义：} \quad [\![e]\!] : \text{state} \to \mathbb{Z}$$
$$\text{新指称语义：} \quad [\![e]\!] : \text{state} \to \mathbb{Z}_{2^{64}}$$

```
Definition add_sem (D1 D2: state -> int64) s: int64 :=
  Int64.add (D1 s) (D2 s).
```

```
Definition sub_sem (D1 D2: state -> int64) s: int64 :=
  Int64.sub (D1 s) (D2 s).
```

```
Definition mul_sem (D1 D2: state -> int64) s: int64 :=
  Int64.mul (D1 s) (D2 s).
```

```
Definition const_sem (n: Z) (s: state): int64 :=
  Int64.repr n.
```

```
Definition var_sem (X: var_name) (s: state): int64 :=
  s X.
```

```
Fixpoint eval_expr_int (e: expr_int) : state -> int64 :=
  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.
```

2 将运算越界定义为表达式求值错误 - 用部分函数定义

在讲解并实现简单解释器之前，我们曾经约定 while 语言的算术运算语义基本遵循 C 标准的规定。特别的，有符号 64 位整数的运算越界应当被视为程序错误（C11 标准 6.5 章节第 5 段落）。然而，这一点并未在上面定义中得到体现。

为了解决这一问题，我们需要能够在定义中表达『程序表达式求值出错』这一概念。这在数学上有两种常见方案。其一是将求值结果由 64 位整数集合改为 64 位整数或求值失败。

- 原指称语义：$\forall e.\quad [\![e]\!] : \text{prog\_state} \to \mathbb{Z}_{2^{64}}$
- 新指称语义：$\forall e.\quad [\![e]\!] : \text{prog\_state} \to \mathbb{Z}_{2^{64}} \cup \{\epsilon\}$
- $[\![n]\!](s) = n$，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 $-2^{63} \leqslant [\![e_1]\!](s) + [\![e_2]\!](s) \leqslant 2^{63} - 1$

  $[\![e_1 + e_2]\!](s) = [\![e_1]\!](s) + [\![e_2]\!](s)$ 三个条件
- 若否

  $[\![e_1 + e_2]\!](s) = \epsilon$
- $[\![e_1 - e_2]\!](s) = \ldots$
- $[\![e_1 * e_2]\!](s) = \ldots$

在 Coq 中可以使用`option`类型描述相关概念。

```
Print option.
```

具体而言，对于任意 Coq 类型`A`，`option A`的元素要么是`Some a`（其中`a`是`A`的元素）要么是`None`。可以看到`option`也是一个 Coq 归纳类型，只不过其定义中并不需要对自身归纳。我们可以像处理先前其他归纳类型一样使用 Coq 中的`match`定义相关的函数或性质，例如：

```
Definition option_plus1 (o: option Z): option Z :=
  match o with
  | Some x => Some (x + 1)
  | None => None
  end.
```

例如上面这一定义说的是：如果`o`的值是`None`那么就返回`None`，如果`o`的值是某个整数（`Some`的情况），那就将它加一返回。

利用类似`option`类型可以改写表达式的语义定义。

```
Definition const_sem (n: Z) (s: state): option int64 :=
  if (n <=? Int64.max_signed) && (n >=? Int64.min_signed)
  then Some (Int64.repr n)
  else None.
```

```
Definition var_sem (x: var_name) (s: state): option int64 :=
  Some (s x).
```

```
Definition add_sem (D1 D2: state -> option int64) (s: state): option int64 :=
  match D1 s, D2 s with
  | Some i1, Some i2 =>
      if (Int64.signed i1 + Int64.signed i2 <=? Int64.max_signed) &&
         (Int64.signed i1 + Int64.signed i2 >=? Int64.min_signed)
      then Some (Int64.add i1 i2)
      else None
  | _, _ => None
  end.
```

```
Definition sub_sem (D1 D2: state -> option int64) (s: state): option int64 :=
  match D1 s, D2 s with
  | Some i1, Some i2 =>
      if (Int64.signed i1 - Int64.signed i2 <=? Int64.max_signed) &&
         (Int64.signed i1 - Int64.signed i2 >=? Int64.min_signed)
      then Some (Int64.sub i1 i2)
      else None
  | _, _ => None
  end.
```

```
Definition mul_sem (D1 D2: state -> option int64) (s: state): option int64 :=
  match D1 s, D2 s with
  | Some i1, Some i2 =>
      if (Int64.signed i1 * Int64.signed i2 <=? Int64.max_signed) &&
         (Int64.signed i1 * Int64.signed i2 >=? Int64.min_signed)
      then Some (Int64.mul i1 i2)
      else None
  | _, _ => None
  end.
```

最终，整数类型表达式的语义可以归结为下面递归定义。

```
Fixpoint eval_expr_int (e: expr_int): state -> option int64 :=
  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.
```

上述定义中除了用到 Coq 的`option`类型，还用到了 Coq 的`bool`类型。

```
Print bool.
```

根据定义`bool`类型只有两种可能的取值：`true`与`false`。请注意，Coq 中的`bool`类型与命题（`Prop`类型）并不相同，前者专门用于关于真与假的真值运算，而后者则可以描述关于任意、存在等真假难以判定、无法计算真值的命题。上面的`eeval`定义要用于计算出`option int64`类型的结果，因此就只能使用`bool`类型的 Coq 函数来做判定了，他们分别是`<=?`, `>=?`与`&&`。

```
Check 1 <=? 2.
Check 1 <= 2.
```

可以看到，`1 <=? 2`是用`bool`类型表达的判断两数大小的结果，这一符号对应的定义是：`Z.leb 1 2`。而`1 <= 2`则是关于两数大小关系的命题，这一符号对应的定义是：`Z.le 1 2`。Coq 标准库中也证明了两者的联系。

```
Check Z.leb_le.
```

这个定理说的是：`forall n m : Z, (n <=? m) = true <-> n <= m`。当然，Coq 标准库中还有很多类似的有用的性质，这里就不再一一罗列了，相关信息也可以通过`Search Z.leb`或`Search Z.le`等方法查找。

最后，在 Coq 中，可以用`&&`表示布尔值的合取（Coq 定义是`andb`）并且使用`if`，`then`，`else`针对布尔表达式求值为真以及为假的情况分别采取不同的求值方法。将这些定义组合在一起，就得到了上面的`eval_expr_int`定义。

下面是一些证明自动化指令。

```
Ltac int64_arith_simpl :=
  repeat
  match goal with
  | |- context [if ?X then Some ?A else None] =>
         match X with
         | context [?A <=? ?B] =>
            first
            [ assert (A <= B) as HHH by int64_lia;
              apply Z.leb_le in HHH;
              rewrite HHH; clear HHH]
         | context [?A >=? ?B] =>
            first
            [ assert (B <= A) as HHH by int64_lia;
              apply Z.geb_le in HHH;
              rewrite HHH; clear HHH]
         | context [true && ?b] => change (true && b) with b
         | context [false && ?b] => change (false && b) with false
         end
  | |- context [if true then ?A else ?B] =>
         change (if true then A else B) with A
  | |- context [if false then ?A else ?B] =>
         change (if false then A else B) with B
  | |- context [match Some ?A with | Some _ => _ | None => _ end] =>
         cbv beta iota
  | |- context [Int64.signed (Int64.repr ?n)] =>
         rewrite (Int64.signed_repr n) by int64_lia
  end.
```

我们可以在 Coq 中证明一些关于表达式指称语义的基本性质。

```
Example eval_expr_int_fact0: forall st,
  st "x" = Int64.repr 0 ->
  eval_expr_int [["x" + 1]] st = Some (Int64.repr 1).
Proof.
  intros.
  simpl.
  unfold add_sem, var_sem, const_sem.
  rewrite H.
  int64_arith_simpl.
  reflexivity.
Qed.
```

上面定义中有三个分支的定义是相似，我们也可以统一定义。

这里，`Zfun: Z -> Z -> Z`可以看做对整数加法（`Z.add`）、整数减法（`Z.sub`）与整数乘法（`Z.mul`）的抽象。而`i64fun: int64 -> int64 -> int64`可以看做对 64 位整数二元运算的抽象。

```
Definition arith_sem
             (Zfun: Z -> Z -> Z)
             (int64fun: int64 -> int64 -> int64)
             (D1 D2: state -> option int64)                超出运算范围的结果
             (s: state): option int64 :=
  match D1 s, D2 s with
  | Some i1, Some i2 =>
      if (Zfun (Int64.signed i1) (Int64.signed i2)
                               <=? Int64.max_signed) &&
         (Zfun (Int64.signed i1) (Int64.signed i2)
                               >=? Int64.min_signed)
      then Some (int64fun i1 i2)
      else None
  | _, _ => None
  end.
```

例如，下面将`Zfun`取`Z.add`、`int64fun`取`Int64.add`代入：

```
Example arith_sem_add_fact: forall D1 D2 s,
  arith_sem Z.add Int64.add D1 D2 s =
  match D1 s, D2 s with
  | Some i1, Some i2 =>
      if (Int64.signed i1 + Int64.signed i2
                               <=? Int64.max_signed) &&
         (Int64.signed i1 + Int64.signed i2
                               >=? Int64.min_signed)
      then Some (Int64.add i1 i2)
      else None
  | _, _ => None
  end.
Proof. intros. reflexivity. Qed.
```

这样，`eeval`的定义就可以简化为：

```
Fixpoint eval_expr_int (e: expr_int):
  state -> option int64 :=
  match e with
  | EConst n =>
      const_sem n
  | EVar X =>
      var_sem X
  | EAdd e1 e2 =>
      arith_sem Z.add Int64.add
        (eval_expr_int e1) (eval_expr_int e2)
  | ESub e1 e2 =>
      arith_sem Z.sub Int64.sub
        (eval_expr_int e1) (eval_expr_int e2)
  | EMul e1 e2 =>
      arith_sem Z.mul Int64.mul
        (eval_expr_int e1) (eval_expr_int e2)
  end.
```

3 将运算越界定义为表达式求值错误 - 用二元关系定义

更常见的是用这种

下面定义 64 位整数之间在有符号 64 位整数范围内的运算关系。

```
Definition arith_compute1_nrm
             (Zfun: Z -> Z -> Z)
             (i1 i2 i: int64): Prop :=
  let res := Zfun (Int64.signed i1) (Int64.signed i2) in
    i = Int64.repr res /\
    Int64.min_signed <= res <= Int64.max_signed.
```

```
Definition arith_compute1_err
             (Zfun: Z -> Z -> Z)
             (i1 i2: int64): Prop :=
  let res := Zfun (Int64.signed i1) (Int64.signed i2) in
    res < Int64.min_signed \/ res > Int64.max_signed.
```

接下去利用表达式与 64 位整数值间的二元关系表达『程序表达式求值出错』这一概念。具体而言，如果表达式`e`在程序状态`s`上能成果求值且求值结果为`i`，那么`s`与`i`这个有序对就在`e`的语义中。

下面定义统一刻画了三种算术运算的语义。

```
Definition arith_sem1_nrm
             (Zfun: Z -> Z -> Z)
             (D1 D2: state -> int64 -> Prop)
             (s: state)
             (i: int64): Prop :=
  exists i1 i2,
    D1 s i1 /\ D2 s i2 /\
    arith_compute1_nrm Zfun i1 i2 i.
```

```
Definition arith_sem1_err
             (Zfun: Z -> Z -> Z)
             (D1 D2: state -> int64 -> Prop)
             (s: state): Prop :=
  exists i1 i2,
    D1 s i1 /\ D2 s i2 /\
    arith_compute1_err Zfun i1 i2.
```

一个表达式的语义分为两部分：求值成功的情况与求值出错的情况。

```
                                          程序状态和64位整数的二元关系。
Record denote: Type := {
  nrm: state -> int64 -> Prop;
  err: state -> Prop; 求值出错的状态的集合。
}.
```

Coq 中的`Record`与许多程序语言中的结构体是类似的。在上面定义中，每个表达式的语义`D: denote`都有两个域：`D.(nrm)`与`D.(err)`分别描述前面提到的两种情况。

```
Definition arith_sem1 Zfun (D1 D2: denote): denote :=
  {|
    nrm := arith_sem1_nrm Zfun D1.(nrm) D2.(nrm);
    err := D1.(err) ∪ D2.(err) ∪
           arith_sem1_err Zfun D1.(nrm) D2.(nrm);
  |}.
```

```
Definition const_sem (n: Z): denote :=
  {|
    nrm := fun s i =>
             i = Int64.repr n /\
             Int64.min_signed <= n <= Int64.max_signed;
    err := fun s =>
             n < Int64.min_signed \/
             n > Int64.max_signed;
  |}.
```

```
Definition var_sem (X: var_name): denote :=
  {|
    nrm := fun s i => i = s X;
    err := ∅;
  |}.
```

最终，整数类型表达式的语义可以归结为下面递归定义。

```
Fixpoint eval_expr_int (e: expr_int): denote :=
  match e with
  | EConst n =>
      const_sem n
  | EVar X =>
      var_sem X
  | EAdd e1 e2 =>
      arith_sem1 Z.add (eval_expr_int e1) (eval_expr_int e2)
  | ESub e1 e2 =>
      arith_sem1 Z.sub (eval_expr_int e1) (eval_expr_int e2)
  | EMul e1 e2 =>
      arith_sem1 Z.mul (eval_expr_int e1) (eval_expr_int e2)
  end.
```

4 未初始化的变量

在 C 语言和很多实际编程语言中，都不允许或不建议在运算中或赋值中使用未初始化的变量的值。若要根据这一设定定义程序语义，那么就需要修改程序状态的定义。变量的值可能是一个 64 位整数，也可能是未初始化。

```
Inductive val: Type :=
| Vuninit: val
| Vint (i: int64): val.
```

程序状态就变成变量名到`val`的函数。

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

表达式的指称依旧包含原有的两部分。

```
Record denote: Type := {
  nrm: state -> int64 -> Prop;
  err: state -> Prop;
}.
```

唯有整数类型表达式中变量的语义需要重新定义。

```
Definition var_sem (X: var_name): denote :=
  {|
    nrm := fun s i => s X = Vint i;
    err := fun s => s X = Vuninit;
  |}.
```

最终，整数类型表达式的语义还是可以写成同样的递归定义。

```
Fixpoint eval_expr_int (e: expr_int): denote :=
  match e with
  | EConst n =>
      const_sem n
  | EVar X =>
      var_sem X
  | EAdd e1 e2 =>
      arith_sem1 Z.add (eval_expr_int e1) (eval_expr_int e2)
  | ESub e1 e2 =>
      arith_sem1 Z.sub (eval_expr_int e1) (eval_expr_int e2)
  | EMul e1 e2 =>
      arith_sem1 Z.mul (eval_expr_int e1) (eval_expr_int e2)
  end.
```

5 While 语言的语义

有了上面这些准备工作，我们可以定义完整 While 语言的语义。完整 While 语言中支持更多运算符，要描述除法和取余运算符的行为，要定义不同于加减乘的运算关系。下面定义参考了 C 标准对于有符号整数除法和取余的规定。

```
Definition arith_compute2_nrm
             (int64fun: int64 -> int64 -> int64)
             (i1 i2 i: int64): Prop :=
  i = int64fun i1 i2 /\
  Int64.signed i2 <> 0 /\
  (Int64.signed i1 <> Int64.min_signed \/
   Int64.signed i2 <> - 1).
```

```
Definition arith_compute2_err (i1 i2: int64): Prop :=
  Int64.signed i2 = 0 \/
  (Int64.signed i1 = Int64.min_signed /\
   Int64.signed i2 = - 1).
```

下面定义的比较运算关系利用了 CompCert 库定义的`comparison`类型和`Int64.cmp`函数。

```
Definition cmp_compute_nrm
             (c: comparison)
             (i1 i2 i: int64): Prop :=
  i = if Int64.cmp c i1 i2
      then Int64.repr 1
      else Int64.repr 0.
```

一元运算的行为比较容易定义：

```
Definition neg_compute_nrm (i1 i: int64): Prop :=
  i = Int64.neg i1 /\
  Int64.signed i1 <> Int64.min_signed.
```

```
Definition neg_compute_err (i1: int64): Prop :=
  Int64.signed i1 = Int64.min_signed.
```

```
Definition not_compute_nrm (i1 i: int64): Prop :=
  Int64.signed i1 <> 0 /\ i = Int64.repr 0 \/
  i1 = Int64.repr 0 /\ i = Int64.repr 1.
```

最后，二元布尔运算的行为需要考虑短路求值的情况。下面定义中的缩写`sc`表示 short circuit。

```
Definition SC_and_compute_nrm (i1 i: int64): Prop :=
  i1 = Int64.repr 0 /\ i = Int64.repr 0.
```

```
Definition SC_or_compute_nrm (i1 i: int64): Prop :=
  Int64.signed i1 <> 0 /\ i = Int64.repr 1.
```

```
Definition NonSC_and (i1: int64): Prop :=
  Int64.signed i1 <> 0.
```

```
Definition NonSC_or (i1: int64): Prop :=
  i1 = Int64.repr 0.
```

```
Definition NonSC_compute_nrm (i2 i: int64): Prop :=
  i2 = Int64.repr 0 /\ i = Int64.repr 0 \/
  Int64.signed i2 <> 0 /\ i = Int64.repr 1.
```

程序状态依旧是变量名到 64 位整数或未初始化值的函数，表达式的指称依旧包含成功求值情况与求值失败情况这两部分。

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

```
Record EDenote: Type := {
  nrm: state -> int64 -> Prop;
  err: state -> Prop;
}.
```

各运算符语义的详细定义见 Coq 代码。

所有运算符的语义中，二元布尔运算由于涉及短路求值，其定义是最复杂的。

```
Definition and_sem_nrm
             (D1 D2: state -> int64 -> Prop)
             (s: state)
             (i: int64): Prop :=
  exists i1,
    D1 s i1 /\
    (SC_and_compute_nrm i1 i \/
     NonSC_and i1 /\
     exists i2,
       D2 s i2 /\ NonSC_compute_nrm i2 i).
```

```
Definition and_sem_err
             (D1: state -> int64 -> Prop)
             (D2: state -> Prop)
             (s: state): Prop :=
  exists i1,
    D1 s i1 /\ NonSC_and i1 /\ D2 s.
```

```
Definition and_sem (D1 D2: EDenote): EDenote :=
  {|
    nrm := and_sem_nrm D1.(nrm) D2.(nrm);
    err := D1.(err) ∪ and_sem_err D1.(nrm) D2.(err);
  |}.
```

```
Definition or_sem_nrm
             (D1 D2: state -> int64 -> Prop)
             (s: state)
             (i: int64): Prop :=
  exists i1,
    D1 s i1 /\
    (SC_or_compute_nrm i1 i \/
     NonSC_or i1 /\
     exists i2,
       D2 s i2 /\ NonSC_compute_nrm i2 i).
```

```
Definition or_sem_err
             (D1: state -> int64 -> Prop)
             (D2: state -> Prop)
             (s: state): Prop :=
  exists i1,
    D1 s i1 /\ NonSC_or i1 /\ D2 s.
```

```
Definition or_sem (D1 D2: EDenote): EDenote :=
  {|
    nrm := or_sem_nrm D1.(nrm) D2.(nrm);
    err := D1.(err) ∪ or_sem_err D1.(nrm) D2.(err);
  |}.
```

最终我们可以将所有一元运算与二元运算的语义汇总起来：

```
Definition unop_sem (op: unop) (D: EDenote): EDenote :=
  match op with
  | ONeg => neg_sem D
  | ONot => not_sem D
  end.
```

```
Definition binop_sem (op: binop) (D1 D2: EDenote): EDenote :=
  match op with
  | OOr => or_sem D1 D2
  | OAnd => and_sem D1 D2
  | OLt => cmp_sem Clt D1 D2
  | OLe => cmp_sem Cle D1 D2
  | OGt => cmp_sem Cgt D1 D2
  | OGe => cmp_sem Cge D1 D2
  | OEq => cmp_sem Ceq D1 D2
  | ONe => cmp_sem Cne D1 D2
  | OPlus => arith_sem1 Z.add D1 D2
  | OMinus => arith_sem1 Z.sub D1 D2
  | OMul => arith_sem1 Z.mul D1 D2
  | ODiv => arith_sem2 Int64.divs D1 D2
  | OMod => arith_sem2 Int64.mods D1 D2
  end.
```

最后补上常数和变量的语义即可得到完整的表达式语义。

```
Fixpoint eval_expr (e: expr): EDenote :=
  match e with
  | EConst n =>
      const_sem n
  | EVar X =>
      var_sem X
  | EBinop op e1 e2 =>
      binop_sem op (eval_expr e1) (eval_expr e2)
  | EUnop op e1 =>
      unop_sem op (eval_expr e1)
  end.
```

基于表达式的指称语义，可以证明一些简单性质。

```
Lemma const_plus_const_nrm:
  forall (n m: Z) (s: state) (i: int64),
    (eval_expr (EBinop OPlus (EConst n) (EConst m))).(nrm) s i ->
    (eval_expr (EConst (n + m))).(nrm) s i.
(* 证明详见Coq源代码。 *)
```

下面定义程序语句的语义。程序语句的语义包含三种情况：正常运行终止、运行出错以及安全运行但不终止。

```
Record CDenote: Type := {
  nrm: state -> state -> Prop;
  err: state -> Prop;
  inf: state -> Prop
}.
```

空语句的语义：

```
Definition skip_sem: CDenote :=
  {|
    nrm := Rels.id;
    err := ∅;
    inf := ∅;
  |}.
```

赋值语句的语义：

```
Definition asgn_sem
             (X: var_name)
             (D: EDenote): CDenote :=
  {|
    nrm := fun s1 s2 =>
      exists i,
        D.(nrm) s1 i /\ s2 X = Vint i /\
        (forall Y, X <> Y -> s2 Y = s1 Y);
    err := D.(err);
    inf := ∅;
  |}.
```

顺序执行语句的语义：

```
Definition seq_sem (D1 D2: CDenote): CDenote :=
  {|
    nrm := D1.(nrm) ○ D2.(nrm);
    err := D1.(err) ∪ (D1.(nrm) ○ D2.(err));
    inf := D1.(inf) ∪ (D1.(nrm) ○ D2.(inf));
  |}.
```

条件分支语句的语义：

```
Definition test_true (D: EDenote):
  state -> state -> Prop :=
  Rels.test
    (fun s =>
       exists i, D.(nrm) s i /\ Int64.signed i <> 0).
```

```
Definition test_false (D: EDenote):
  state -> state -> Prop :=
  Rels.test (fun s => D.(nrm) s (Int64.repr 0)).
```

```
Definition if_sem
             (D0: EDenote)
             (D1 D2: CDenote): CDenote :=
  {|
    nrm := (test_true D0 ○ D1.(nrm)) ∪
           (test_false D0 ○ D2.(nrm));
    err := D0.(err) ∪
           (test_true D0 ○ D1.(err)) ∪
           (test_false D0 ○ D2.(err));
    inf := (test_true D0 ○ D1.(inf)) ∪
           (test_false D0 ○ D2.(inf))
  |}.
```

二元关系和一元关系的复合。

在 while 语句的语义定义中，程序正常运行终止的情况与程序运行出错终止的情况可以通过穷举循环体迭代运行次数的方式来定义。

而 while 循环语句不终止的情况又分为两种：每次执行循环体程序都正常运行终止但是由于一直满足循环条件将执行无穷多次循环体；某次执行循环体时，执行循环体的过程本身不终止。下面定义的`is_inf`描述了以下关于程序状态集合`X`的性质：从集合`X`中的任意一个状态出发，计算循环条件的结果都为真（也不会计算出错），进入循环体执行后，要么正常运行终止并且终止于另一个（可以是同一个）`X`集合中的状态上，要么循环体运行不终止。

```
Definition is_inf
             (D0: EDenote)
             (D1: CDenote)
             (X: state -> Prop): Prop :=
  X ⊆ test_true D0 ∘ ((D1.(nrm) ∘ X) ∪ D1.(inf)).
```

这样一来，循环正常终止与循环出错终止的行为可以通过对所有迭代次数取并集完成定义。而循环不终止的情况则可以定义为所有满足`is_inf`性质的集合的并集。

```
Definition while_sem
             (D0: EDenote)
             (D1: CDenote): CDenote :=
  {|
    nrm := BW_LFix
             (fun X =>
                test_true D0 ∘ D1.(nrm) ∘ X ∪
                test_false D0);
    err := BW_LFix
             (fun X =>
                test_true D0 ∘ D1.(nrm) ∘ X ∪
                D0.(err));
    inf := Sets.general_union (is_inf D0 D1);
  |}.
```

程序语句的语义可以最后表示成下面递归函数。

```
Fixpoint eval_com (c: com): CDenote :=
  match c with
  | CSkip =>
      skip_sem
  | CAsgn X e =>
      asgn_sem X (eval_expr e)
  | CSeq c1 c2 =>
      seq_sem (eval_com c1) (eval_com c2)
  | CIf e c1 c2 =>
      if_sem (eval_expr e) (eval_com c1) (eval_com c2)
  | CWhile e c1 =>
      while_sem (eval_expr e) (eval_com c1)
  end.
```

6 WhileDeref 语言的语义

程序状态：

```
Record state: Type := {
  vars: var_name -> val;
  mem: int64 -> option val;
}.
```

这里，`s.(vars)`表示程序状态`s`上变量的值，`s.(mem)`表示程序状态`s`上的内存信息，地址用 64 位有符号整数表示，一个地址上的值是`Some`表示有读写权限，`None`表示没有读写权限。

下面定义表达式的指称语义。表达式指称的定义、二元运算、一元运算与常量的语义都与原先相同。

```
Record EDenote: Type := {
  nrm: state -> int64 -> Prop;
  err: state -> Prop;
}.
```

程序表达式为单个变量时的语义需要做少许修改,其语义定义应当使用程序状态中变量的部分:`s.(vars)`。

```
Definition var_sem (X: var_name): EDenote :=
  {|
    nrm := fun s i => s.(vars) X = Vint i;
    err := fun s => s.(vars) X = Vuninit;
  |}.
```

最后，需要新定义『解引用』的语义，其语义定义应当使用程序状态中内存的部分：`s.(mem)`。

```
Definition deref_sem_nrm
             (D1: state -> int64 -> Prop)
             (s: state)
             (i: int64): Prop :=
  exists i1, D1 s i1 /\ s.(mem) i1 = Some (Vint i).
```

```
Definition deref_sem_err
             (D1: state -> int64 -> Prop)
             (s: state): Prop :=
  exists i1,
    D1 s i1 /\
    (s.(mem) i1 = None \/ s.(mem) i1 = Some Vuninit).
```

```
Definition deref_sem (D1: EDenote): EDenote :=
  {|
    nrm := deref_sem_nrm D1.(nrm);
    err := D1.(err) ∪ deref_sem_err D1.(nrm);
  |}.
```

在递归定义的表达式语义中，也需要加上『解引用』的情形。

```
Fixpoint eval_expr (e: expr): EDenote :=
  match e with
  | EConst n =>
      const_sem n
  | EVar X =>
      var_sem X
  | EBinop op e1 e2 =>
      binop_sem op (eval_expr e1) (eval_expr e2)
  | EUnop op e1 =>
      unop_sem op (eval_expr e1)
  | EDeref e1 =>
      deref_sem (eval_expr e1)
  end.
```

在表达式语义定义的基础上，`test_true`与`test_false`的定义不变。

```
Definition test_true (D: EDenote):
  state -> state -> Prop :=
  Rels.test
    (fun s =>
       exists i, D.(nrm) s i /\ Int64.signed i <> 0).
```

```
Definition test_false (D: EDenote):
  state -> state -> Prop :=
  Rels.test (fun s => D.(nrm) s (Int64.repr 0)).
```

程序语句的指称定义不变，依然包括三种情况：正常运行终止、运行出错以及运行不终止。空语句、顺序执行、条件分支语句与 while 循环语句的语义定义也不变。

```
Record CDenote: Type := {
  nrm: state -> state -> Prop;
  err: state -> Prop;
  inf: state -> Prop
}.
```

需要新定义的是两种赋值语句的语义。变量赋值语句的语义与原赋值语句的语义基本相同，但需要额外说明对变量赋值不修改内存上的值。

```
Definition asgn_var_sem
             (X: var_name)
             (D: EDenote): CDenote :=
  {|
    nrm := fun s1 s2 =>
      exists i,
        D.(nrm) s1 i /\ s2.(vars) X = Vint i /\
        (forall Y, X <> Y -> s2.(vars) Y = s1.(vars) Y) /\
        (forall p, s1.(mem) p = s2.(mem) p);
    err := D.(err);
    inf := ∅;
  |}.
```

向某地址赋值的语义是根据表达式修改内存上的值而不改变任意一个程序变量的值。

```
Definition asgn_deref_sem_nrm
             (D1 D2: state -> int64 -> Prop)
             (s1 s2: state): Prop :=
  exists i1 i2,
    D1 s1 i1 /\
    D2 s1 i2 /\
    s1.(mem) i1 <> None /\
    s2.(mem) i1 = Some (Vint i2) /\
    (forall X, s1.(vars) X = s2.(vars) X) /\
    (forall p, i1 <> p -> s1.(mem) p = s2.(mem) p).
```

```
Definition asgn_deref_sem_err
             (D1: state -> int64 -> Prop)
             (s1: state): Prop :=
  exists i1,
    D1 s1 i1 /\
    s1.(mem) i1 = None.
```

```
Definition asgn_deref_sem
             (D1 D2: EDenote): CDenote :=
  {|
    nrm := asgn_deref_sem_nrm D1.(nrm) D2.(nrm);
    err := D1.(err) ∪ D2.(err) ∪
           asgn_deref_sem_err D2.(nrm);
    inf := ∅;
  |}.
```

在递归定义的程序语句语义中，要加上这两种赋值语句的语义。

```
Fixpoint eval_com (c: com): CDenote :=
  match c with
  | CSkip =>
      skip_sem
  | CAsgnVar X e =>
      asgn_var_sem X (eval_expr e)
  | CAsgnDeref e1 e2 =>
      asgn_deref_sem (eval_expr e1) (eval_expr e2)
  | CSeq c1 c2 =>
      seq_sem (eval_com c1) (eval_com c2)
  | CIf e c1 c2 =>
      if_sem (eval_expr e) (eval_com c1) (eval_com c2)
  | CWhile e c1 =>
      while_sem (eval_expr e) (eval_com c1)
  end.
```

7 WhileD 语言的语义

程序状态：

```
Record state: Type := {
  env: var_name -> int64;
  mem: int64 -> option val;
}.
```

由于表达式中存在取地址操作，我们无法继续沿用原先定义的表达式指称。

『解引用』表达式既可以用作右值也可以用作左值。其作为右值是的语义就是原先我们定义的『解引用』语义。

```
Definition deref_sem_nrm
             (D1: state -> int64 -> Prop)
             (s: state)
             (i: int64): Prop :=
  exists i1, D1 s i1 /\ s.(mem) i1 = Some (Vint i).
```

```
Definition deref_sem_err
             (D1: state -> int64 -> Prop)
             (s: state): Prop :=
  exists i1,
    D1 s i1 /\
    (s.(mem) i1 = None \/ s.(mem) i1 = Some Vuninit).
```

```
Definition deref_sem_r (D1: EDenote): EDenote :=
  {|
    nrm := deref_sem_nrm D1.(nrm);
    err := D1.(err) ∪ deref_sem_err D1.(nrm);
  |}.
```

当程序表达式为单个变量时，它也可以同时用作左值或右值。下面先定义其作为左值时的存储地址。

```
Definition var_sem_l (X: var_name): EDenote :=
  {|
    nrm := fun s i => s.(env) X = i;
    err := ∅;
  |}.
```

基于此，可以又定义它作为右值时的值。

```
Definition var_sem_r (X: var_name): EDenote :=
  deref_sem_r (var_sem_l X).
```

最后，我们可以用互递归函数（mutually recursive function）在 Coq 中定义表达式的语义。需注意，解引用表达式`* e`作为左值时的存储地址就是`e`作为右值时的值，而取地址表达式`& e`作为右值时的值就是`e`作为左值时的存储地址。这都不需要在额外定义。同时，常量、一元运算、二元运算、取地址表达式这些都不能用作左值表达式。

```
Fixpoint eval_r (e: expr): EDenote :=
  match e with
  | EConst n =>
      const_sem n
  | EVar X =>
      var_sem_r X
  | EBinop op e1 e2 =>
      binop_sem op (eval_r e1) (eval_r e2)
  | EUnop op e1 =>
      unop_sem op (eval_r e1)
  | EDeref e1 =>
      deref_sem_r (eval_r e1)
  | EAddrOf e1 =>
      eval_l e1
  end
```

```
with eval_l (e: expr): EDenote :=
  match e with
  | EVar X =>
      var_sem_l X
  | EDeref e1 =>
      eval_r e1
  | _ =>
      {| nrm := ∅; err := Sets.full; |}
  end.
```

这里`test_true`与`test_false`的定义不变，不过之后只会将其作用在表达式的右值上。

程序语句的指称定义不变，依然包括三种情况：正常运行终止、运行出错以及运行不终止。空语句、顺序执行、条件分支语句与 while 循环语句的语义定义也不变。

向地址赋值的语义与原先定义基本相同，只是现在需要规定所有变量的地址不被改变，而非所有变量的值不被改变。

```
Definition asgn_deref_sem_nrm
             (D1 D2: state -> int64 -> Prop)
             (s1 s2: state): Prop :=
  exists i1 i2,
    D1 s1 i1 /\
    D2 s1 i2 /\
    s1.(mem) i1 <> None /\
    s2.(mem) i1 = Some (Vint i2) /\
    (forall X, s1.(env) X = s2.(env) X) /\
    (forall p, i1 <> p -> s1.(mem) p = s2.(mem) p).
```

```
Definition asgn_deref_sem_err
             (D1: state -> int64 -> Prop)
             (s1: state): Prop :=
  exists i1,
    D1 s1 i1 /\
    s1.(mem) i1 = None.
```

```
Definition asgn_deref_sem
             (D1 D2: EDenote): CDenote :=
  {|
    nrm := asgn_deref_sem_nrm D1.(nrm) D2.(nrm);
    err := D1.(err) ∪ D2.(err) ∪
           asgn_deref_sem_err D2.(nrm);
    inf := ∅;
  |}.
```

变量赋值的行为可以基于此定义。

```
Definition asgn_var_sem
             (X: var_name)
             (D1: EDenote): CDenote :=
  asgn_deref_sem (var_sem_l X) D1.
```

在递归定义的程序语句语义中，只会直接使用表达式用作右值时的值。

```
Fixpoint eval_com (c: com): CDenote :=
  match c with
  | CSkip =>
      skip_sem
  | CAsgnVar X e =>
      asgn_var_sem X (eval_r e)
  | CAsgnDeref e1 e2 =>
      asgn_deref_sem (eval_r e1) (eval_r e2)
  | CSeq c1 c2 =>
      seq_sem (eval_com c1) (eval_com c2)
  | CIf e c1 c2 =>
      if_sem (eval_r e) (eval_com c1) (eval_com c2)
  | CWhile e c1 =>
      while_sem (eval_r e) (eval_com c1)
  end.
```

8 WhileDC 语言的语义

由于在程序语句中加入了控制流，程序语句的指称定义需要做相应改变。

```
Record CDenote: Type := {
  nrm: state -> state -> Prop;
  brk: state -> state -> Prop;
  cnt: state -> state -> Prop;
  err: state -> Prop;
  inf: state -> Prop
}.
```

具体定义见 Coq 代码。
空语句的语义

```
Definition skip_sem: CDenote :=
  {|
    nrm := Rels.id;
    brk := ∅;
    cnt := ∅;
    err := ∅;
    inf := ∅;
  |}.
```

Break 语句的语义

```
Definition brk_sem: CDenote :=
  {|
    nrm := ∅;
    brk := Rels.id;
    cnt := ∅;
    err := ∅;
    inf := ∅;
  |}.
```

Continue 语句的语义

```
Definition cnt_sem: CDenote :=
  {|
    nrm := ∅;
    brk := ∅;
    cnt := Rels.id;
    err := ∅;
    inf := ∅;
  |}.
```

顺序执行语句的语义

```
Definition seq_sem (D1 D2: CDenote): CDenote :=
  {|
    nrm := D1.(nrm) ∘ D2.(nrm);
    brk := D1.(brk) ∪ (D1.(nrm) ∘ D2.(brk));
    cnt := D1.(cnt) ∪ (D1.(nrm) ∘ D2.(cnt));
    err := D1.(err) ∪ (D1.(nrm) ∘ D2.(err));
    inf := D1.(inf) ∪ (D1.(nrm) ∘ D2.(inf));
  |}.
```

If 语句的语义

```
Definition if_sem
             (D0: EDenote)
             (D1 D2: CDenote): CDenote :=
  {|
    nrm := (test_true D0 ∘ D1.(nrm)) ∪
           (test_false D0 ∘ D2.(nrm));
    brk := (test_true D0 ∘ D1.(brk)) ∪
           (test_false D0 ∘ D2.(brk));
    cnt := (test_true D0 ∘ D1.(cnt)) ∪
           (test_false D0 ∘ D2.(cnt));
    err := D0.(err) ∪
           (test_true D0 ∘ D1.(err)) ∪
           (test_false D0 ∘ D2.(err));
    inf := (test_true D0 ∘ D1.(inf)) ∪
           (test_false D0 ∘ D2.(inf))
  |}.
```

其余语句的语义定义见 Coq 代码。