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

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。

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

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.
```

3 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.
```

4 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 (D1 D2: EDenote): EDenote :=
  {|
    nrm := and_sem_nrm D1.(nrm) D2.(nrm);
    err := D1.(err) ∪ 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 (D1 D2: EDenote): EDenote :=
  {|
    nrm := or_sem_nrm D1.(nrm) D2.(nrm);
    err := D1.(err) ∪ 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_true D0 ∘ D2.(err));
    inf := (test_true D0 ∘ D1.(inf)) ∪
           (test_true D0 ∘ D2.(inf))
  |}.
```

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。

```
Fixpoint iter_nrm_lt_n
           (D0: EDenote)
           (D1: CDenote)
           (n: nat):
  state -> state -> Prop :=
  match n with
  | O => ∅
  | S n0 =>
      (test_true D0 ∘ D1.(nrm) ∘ iter_nrm_lt_n D0 D1 n0) ∪
      (test_false D0)
  end.
```

```
Fixpoint iter_err_lt_n
           (D0: EDenote)
           (D1: CDenote)
           (n: nat): state -> Prop :=
  match n with
  | O => ∅
  | S n0 =>
     (test_true D0 ∘
        (D1.(nrm) ∘ iter_err_lt_n D0 D1 n0) ∪
        (D1.(err))) ∪
      D0.(err)
  end.
```

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 := ⋃ (iter_nrm_lt_n D0 D1);
    err := ⋃ (iter_err_lt_n D0 D1);
    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.
```