现实程序语言的指称语义

表示 64 位整数运算的整数表达式语义

原程序状态：

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

新程序状态：

$$\text{state} \triangleq \text{var\_name} \to \mathbb{Z}_{2^{64}}$$

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

64 位整数运算的例子

- `Int64.add` 表示 64 位整数的加法：

```
Check Int64.add.
```

Coq 返回：`Int64.add : int64 -> int64 -> int64`

64 位整数运算的例子

- `Int64.add` 表示 64 位整数的加法：

```
Check Int64.add.
```

Coq 返回： `Int64.add : int64 -> int64 -> int64`

- `Int64.and` 表示 64 位整数的按位合取：

```
Check Int64.and.
```

Coq 返回： `Int64.and : int64 -> int64 -> int64`

64 位整数相关的常用函数与常数

- `Int64.repr`
- `Int64.signed`
- `Int64.unsigned`
- `Int64.max_signed`
- `Int64.min_signed`
- `Int64.max_unsigned`

原指称语义：

$$\forall e. \quad [\![e]\!] : \mathrm{state} \rightarrow \mathbb{Z}$$

新指称语义：

$$\forall e. \quad [\![e]\!] : \mathrm{state} \rightarrow \mathbb{Z}_{2^{64}}$$

```
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.
```

将运算越界定义为表达式求值错误

```
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.
```

```
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.
```

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

```
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 := ∅;
  |}.
```

```
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.
```

未初始化的变量

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

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

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

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

While 语言的语义

```
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).
```

```
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.
```

```
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.
```

```
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.
```

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

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

```
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。

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

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

```
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 := ∅;
  |}.
```

```
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));
  |}.
```

```
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))
  |}.
```

```
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.
```

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

```
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);
  |}.
```

程序语句的指称语义

```
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.
```

WhileDeref 语言及其语义

表达式的语法树

```
Inductive expr : Type :=
  | EConst (n: Z): expr
  | EVar (x: var_name): expr
  | EBinop (op: binop) (e1 e2: expr): expr
  | EUnop (op: unop) (e: expr): expr
  | EDeref (e: expr): expr.
```

程序语句的语法树

```
Inductive com : Type :=
  | CSkip: com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com.
```

程序状态

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

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

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

二元运算、一元运算与常量的语义都与原先相同。

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

解引用的语义

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

```
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.
```

```
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)).
```

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

```
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 := ∅;
  |}.
```

```
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 := ∅;
  |}.
```

程序语句的指称语义

```
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.
```

WhileD 语言的语义

程序状态

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

语义定义中的新问题

语义定义中的新问题

- 表达式 e 在程序状态 s 上的值不能决定 $\&e$ 的值

语义定义中的新问题

- 表达式 e 在程序状态 s 上的值不能决定 $\&e$ 的值
- 破坏了指称语义的可组合原则

语义定义中的新问题

- 表达式 e 在程序状态 s 上的值不能决定 $\&e$ 的值
- 破坏了指称语义的可组合原则

- 右值
- 左值

```
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);
  |}.
```

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

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

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

```
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.
```

```
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 := ∅;
  |}.
```

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

程序语句的指称语义

```
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.
```

WhileDC 语言的语义

程序语句的语义

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

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。