小步语义

- $(c,k,s) \rightarrow (c',k',s')$ 表示从程序状态 s 出发，当前程序执行位置是 (c,k)（其中 c 表示当前紧接着要执行的程序，focused program，k 表示执行程序 c 的外层程序语句环境），执行一步之后，程序状态变为 s'，程序执行位置变为 (c',k')。如果不存在这样的 (s',c',k') 就说明程序会在这一步运行出错。

1 单步的概念

- 表达式

```
E :: = N | V | -E | E+E | E-E | E*E | E/E | E%E |
       E<E | E<=E | E==E | E!=E | E>=E | E>E |
       E&&E | E||E | !E
```

- 语句

```
C :: = SKIP |
       V = E |
       C; C |
       if (E) then { C } else { C } |
       while (E) do { C }
```

- 后续执行的程序
  - KSeq(c)
  - KWhileCond(e,c)，KWhileBody(e,c)
  - KIf(c_1,c_2)
  - KAsgnVar(x)
  - KBinopL(op,e)，KBinopR(n,op)
  - KUnOp(op)
- 表达式求值的中间状态可以用二元组 (e,k) 表示，而 k 中的每一项都是以下之一：
  - KBinopL$(?,?)$
  - KBinopR$(?,?)$
  - KUnOp$(?)$
- 表达式求值过程的例子：如果 $s(x)=1$ 并且 $s(y)=2$，那么，

toplevel是+号

focue sed

后续计算的记录在evaluation context中

```
(x + y) + 1, [] ->
x + y, [KBinopL(+, 1)] ->
x, [KBinopL(+, y), KBinopL(+, 1)] ->
1, [KBinopL(+, y), KBinopL(+, 1)] ->
y, [KBinopR(1, +), KBinopL(+, 1)] ->
2, [KBinopR(1, +), KBinopL(+, 1)] ->
3, [KBinopL(+, 1)] ->
1, [KBinopR(3, +)] ->
1, [KBinopR(3, +)] ->
4
```

1的求值结果

- 程序运行的中间状态有两种情况，可以用二元组 (e, k) 或 (c, k) 表示，前者表示程序运行中正在对某表达式进行求值。
- 程序运行过程的例子：如果 $s(i) = 0$、$s(r) = 0$ 并且 $s(n) = 10$，那么

```
while (i < n) do { r=r+i; i=i+1 }, []
i < n, [KWhileCond(i < n, r=r+i; i=i+1)] ->
i, [KBinopL(<, n), KWhileCond(i < n, r = r + i; i = i + 1)] ->
0, [KBinopL(<, n), KWhileCond(i < n, r = r + i; i = i + 1)] ->
n, [KBinopR(0, <), KWhileCond(i < n, r = r + i; i = i + 1)] ->
10, [KBinopR(0, <), KWhileCond(i < n, r = r + i; i = i + 1)] ->
1, [KWhileCond(i < n, r = r + i; i = i + 1)] ->
r = r + i; i = i + 1, [KWhileBody(i < n, r = r + i; i = i + 1)] ->
r = r + i, [KSeq(i = i + 1), KWhileBody(i < n, r = r + i; i = i + 1)] ->
...
```

- Coq 定义

```
Inductive expr_ectx: Type :=
| KBinopL (op: binop) (e: expr)
| KBinopR (i: int64) (op: binop)
| KUnop (op: unop).
```

```
Inductive expr_loc: Type :=
| EL_Value (i: int64)
| EL_FocusedExpr (e: expr)
| EL_Cont (el: expr_loc) (k: expr_ectx).
```

Coq 定义

```
Inductive expr_com_ectx: Type :=
| KWhileCond (e: expr) (c: com)
| KIf (c1 c2: com)
| KAsgnVar (x: var_name).
```

```
Inductive com_ectx: Type :=
| KSeq (c: com)
| KWhileBody (e: expr) (c: com).
```

```
Inductive com_loc: Type :=
| CL_Finished
| CL_FocusedCom (c: com)
| CL_ECont (el: expr_loc) (k: expr_com_ectx)
| CL_CCont (cl: com_loc) (k: com_ectx).
```

2 小步语义的定义

- 变量、加法的小步语义
  - 如果 $n = s(x)$，那么 $(x, \epsilon) \rightarrow (n, \epsilon)$ @ s
  - $(e_1 + e_2, \epsilon) \rightarrow (e_1, \mathrm{KBinopL}(+, e_2))$ @ s
  - $(n_1, \mathrm{KBinopL}(+, e_2)) \rightarrow (e_2, \mathrm{KBinopR}(n_1, +))$ @ s
  - 如果 $n = n_1 + n_2$ 并且 $-2^{63} \leqslant n \leqslant 2^{63} - 1$，那么 $(n_2, \mathrm{KBinopR}(n_1, +)) \rightarrow (n, \epsilon)$ @ s
- 加法的小步语义定义需要补充 evaluation context 相关性质一条
  - 如果 $(e, k) \rightarrow (e', k')$ @ s 并且 k_0 具有以下形式之一：
    * KBinopL(?, ?)
    * KBinopR(?, ?)
    * KUnOp(?)

    那么 $(e, k \cdot k_0) \rightarrow (e', k' \cdot k_0)$ @ s
- 短路求值的语义
  - $(e_1 \ op \ e_2, \epsilon) \rightarrow (e_1, \mathrm{KBinopL}(op, e_2))$ @ s
  - 如果 $n_1 = 0$，$(n_1, \mathrm{KBinopL}(op, e_2)) \rightarrow (0, \epsilon)$ @ s
  - 如果 $n_1 \neq 0$，$(n_1, \mathrm{KBinopL}(op, e_2)) \rightarrow (e_2, \mathrm{KBinopR}(n_1, \&\&))$ @ s
- 其他表达式的小步语义是类似的，这里略去。
- Coq 预备定义

```
Definition SC_compute_nrm (op: binop) (i i': int64): Prop :=
  match op with
  | OAnd => SC_and_compute_nrm i i'
  | OOr => SC_or_compute_nrm i i'
  | _ => False
  end.
```

```
Definition NonSC (op: binop) (i: int64): Prop :=
  match op with
  | OAnd => NonSC_and i
  | OOr => NonSC_or i
  | _ => True
  end.
```

```
Definition binop_compute_nrm (op: binop):
  int64 -> int64 -> int64 -> Prop :=
  match op with
  | OOr => fun i1 i2 i => NonSC_compute_nrm i2 i
  | OAnd => fun i1 i2 i => NonSC_compute_nrm i2 i
  | OLt => cmp_compute_nrm Clt
  | OLe => cmp_compute_nrm Cle
  | OGt => cmp_compute_nrm Cgt
  | OGe => cmp_compute_nrm Cge
  | OEq => cmp_compute_nrm Ceq
  | ONe => cmp_compute_nrm Cne
  | OPlus => arith_compute1_nrm Z.add
  | OMinus => arith_compute1_nrm Z.sub
  | OMul => arith_compute1_nrm Z.mul
  | ODiv => arith_compute2_nrm Int64.divs
  | OMod => arith_compute2_nrm Int64.mods
  end.
```

```
Definition unop_compute_nrm (op: unop):
  int64 -> int64 -> Prop :=
  match op with
  | ONeg => neg_compute_nrm
  | ONot => not_compute_nrm
  end.
```

- 表达式求值小步语义的 Coq 定义

```
Inductive estep (s: state):
  expr_loc -> expr_loc -> Prop :=
| ES_Var: forall (x: var_name) (i: int64),
    s x = Vint i ->
    estep s
      (EL_FocusedExpr (EVar x))
      (EL_Value i)
| ES_Const: forall (n: Z),
    n <= Int64.max_signed ->
    n >= Int64.min_signed ->
    estep s
      (EL_FocusedExpr (EConst n))
      (EL_Value (Int64.repr n))
```

```
| ES_BinopL: forall op e1 e2,
    estep s
      (EL_FocusedExpr (EBinop op e1 e2))
      (EL_Cont (EL_FocusedExpr e1) (KBinopL op e2))
| ES_BinopR_SC: forall op n1 n1' e2,
    SC_compute_nrm op n1 n1' ->
    estep s
      (EL_Cont (EL_Value n1) (KBinopL op e2))
      (EL_Value n1')
| ES_BinopR_NSC: forall op n1 e2,
    NonSC op n1 ->
    estep s
      (EL_Cont (EL_Value n1) (KBinopL op e2))
      (EL_Cont (EL_FocusedExpr e2) (KBinopR n1 op))
| ES_BinopStep: forall op n1 n2 n,
    binop_compute_nrm op n1 n2 n ->
    estep s
      (EL_Cont (EL_Value n2) (KBinopR n1 op))
      (EL_Value n)
```

```
| ES_Unop: forall op e,
    estep s
      (EL_FocusedExpr (EUnop op e))
      (EL_Cont (EL_FocusedExpr e) (KUnop op))
| ES_UnopStep: forall op n0 n,
    unop_compute_nrm op n0 n ->
    estep s
      (EL_Cont (EL_Value n0) (KUnop op))
      (EL_Value n)
```

```
| ES_Cont: forall el1 el2 k,
    estep s el1 el2 ->
    estep s (EL_Cont el1 k) (EL_Cont el2 k).
```

- 赋值语句的小步语义
  - $(x = e, \epsilon, s) \to (e, \mathrm{KAsgnVar}(x), s)$
  - 如果
    * $s'(x) = n$，
    * 对于任意 $y \neq x$ 都有 $s'(y) = s(y)$ 并且

    那么 $(n, \mathrm{KAsgnVar}(x), s) \to (\epsilon, \epsilon, s')$
- 赋值语句的小步语义还需要补充下面关于 evaluation context 的性质
  - 如果 $(e, k) \to (e', k')$ @ s 并且 k_0 具有以下形式之一：
    * $\mathrm{KWhileCond}(?, ?)$
    * $\mathrm{KIf}(?, ?)$
    * $\mathrm{KAsgnVar}(?)$，

    那么 $(e, k \cdot k_0, s) \to (e', k' \cdot k_0, s)$
- 顺序执行的小步语义
  - $(c_1; c_2, \epsilon, s) \to (c_1, \mathrm{KSeq}(c_2), s)$
  - $(\epsilon, \mathrm{KSeq}(c), s) \to (c, \epsilon, s)$

- 顺序执行的小步语义需要补充下述关于 evaluation context 的性质
  - 如果 $(e/c, k, s) \rightarrow (e'/c', k', s')$ 并且 k_0 具有以下形式之一：
    - KSeq(?)
    - KWhileBody(?, ?)

    那么 $(e/c, k \cdot k_0, s) \rightarrow (e'/c', k' \cdot k_0, s')$
- 条件分支语句的小步语义
  - $(\text{if } e \text{ then } \{c_1\} \text{ else } \{c_2\}, \epsilon, s) \rightarrow (e, \text{KIf}(c_1, c_2), s)$
  - 如果 $n \neq 0$，$(n, \text{KIf}(c_1, c_2), s) \rightarrow (c_1, \epsilon, s)$
  - $(0, \text{KIf}(c_1, c_2), s) \rightarrow (c_2, \epsilon, s)$
- 循环语句的小步语义
  - $(\text{while } e \text{ do } \{c\}, \epsilon, s) \rightarrow (e, \text{KWhileCond}(e, c), s)$
  - 如果 $n \neq 0$，$(n, \text{KWhileCond}(e, c), s) \rightarrow (c, \text{KWhileBody}(e, c), s)$
  - $(0, \text{KWhileCond}(e, c), s) \rightarrow (\epsilon, \epsilon, s)$
  - $(\epsilon, \text{KWhileBody}(e, c), s) \rightarrow (e, \text{KWhileCond}(e, c), s)$
- 程序运行小步语义的 Coq 定义

```
Inductive cstep:
  com_loc * state -> com_loc * state -> Prop :=
| CS_AsgnVar: forall s x e,
    cstep
      (CL_FocusedCom (CAsgn x e), s)
      (CL_ECont (EL_FocusedExpr e) (KAsgnVar x), s)
| CS_AsgnVarStep: forall s1 s2 x n,
    s2 x = Vint n ->
    (forall y, x <> y -> s2 y = s1 y) ->
    cstep
      (CL_ECont (EL_Value n) (KAsgnVar x), s1)
      (CL_Finished, s2)
```

```
| CS_Seq: forall s c1 c2,
    cstep
      (CL_FocusedCom (CSeq c1 c2), s)
      (CL_CCont (CL_FocusedCom c1) (KSeq c2), s)
| CS_SeqStep: forall s c,
    cstep
      (CL_CCont CL_Finished (KSeq c), s)
      (CL_FocusedCom c, s)
```

```
| CS_If: forall s e c1 c2,
    cstep
      (CL_FocusedCom (CIf e c1 c2), s)
      (CL_ECont (EL_FocusedExpr e) (KIf c1 c2), s)
| CS_IfStepTrue: forall s n c1 c2,
    Int64.signed n <> 0 ->
    cstep
      (CL_ECont (EL_Value n) (KIf c1 c2), s)
      (CL_FocusedCom c1, s)
| CS_IfStepFalse: forall s n c1 c2,
    n = Int64.repr 0 ->
    cstep
      (CL_ECont (EL_Value n) (KIf c1 c2), s)
      (CL_FocusedCom c2, s)
```

```
| CS_WhileBegin: forall s e c,
    cstep
      (CL_FocusedCom (CWhile e c), s)
      (CL_ECont (EL_FocusedExpr e) (KWhileCond e c), s)
| CS_WhileStepTrue: forall s n e c,
    Int64.signed n <> 0 ->
    cstep
      (CL_ECont (EL_Value n) (KWhileCond e c), s)
      (CL_CCont (CL_FocusedCom c) (KWhileBody e c), s)
| CS_WhileStepFalse: forall s n e c,
    n = Int64.repr 0 ->
    cstep
      (CL_ECont (EL_Value n) (KWhileCond e c), s)
      (CL_Finished, s)
| CS_WhileRestart: forall s e c,
    cstep
      (CL_CCont CL_Finished (KWhileBody e c), s)
      (CL_ECont (EL_FocusedExpr e) (KWhileCond e c), s)
```

```
| CS_CSkip: forall s,
    cstep (CL_FocusedCom CSkip, s) (CL_Finished, s)
| CS_ECont: forall el1 el2 s k,
    estep s el1 el2 ->
    cstep (CL_ECont el1 k, s) (CL_ECont el2 k, s)
| CS_CCont: forall cl1 s1 cl2 s2 k,
    cstep (cl1, s1) (cl2, s2) ->
    cstep (CL_CCont cl1 k, s1) (CL_CCont cl2 k, s2).
```

3 多步关系

- 多步关系
  - $(?,?,?) \rightarrow^* (?,?,?)$ 是 $(?,?,?) \rightarrow (?,?,?)$ 的自反传递闭包
  - $(?,?) \rightarrow^* (?,?)$ @ s 是 $(?,?) \rightarrow (?,?)$ @ s 的自反传递闭包
- Coq 中定义多步关系

```
Definition multi_estep (s: state):
  expr_loc -> expr_loc -> Prop :=
  clos_refl_trans (estep s).
```

```
Definition multi_cstep:
  com_loc * state -> com_loc * state -> Prop :=
  clos_refl_trans cstep.
```

- 引理：如果 $(e/c, k, s) \to^* (e'/c', k', s')$ 并且 k_0 具有以下形式：

    - KSeq(?)
    - KWhileBody(?, ?)

  那么 $(e/c, k \cdot k_0, s) \to^* (e'/c', k' \cdot k_0, s')$

- 引理：如果 $(e, k) \to^* (e', k')$ @ s 并且 k_0 具有以下形式：

    - KBinopL(?, ?)
    - KBinopR(?, ?)
    - KUnOp(?)

  那么 $(e, k \cdot k_0) \to^* (e', k' \cdot k_0)$ @ s

- 引理：如果 $(e, k) \to^* (e', k')$ @ s，k 具有以下形式：

    - KWhileCond(?, ?)
    - KIf(?, ?)
    - KAsgnVar(?)

  那么 $(e, k \cdot k_0, s) \to^* (e', k' \cdot k_0, s)$

- 多步关系的基本性质

```
Lemma MES_Cont: forall s el1 el2 k,
  multi_estep s el1 el2 ->
  multi_estep s (EL_Cont el1 k) (EL_Cont el2 k).
Proof.
  intros.
  induction_1n H.
  + reflexivity.
  + transitivity_1n (EL_Cont el0 k).
    - apply ES_Cont; tauto.
    - tauto.
Qed.
```

```
Lemma MCS_ECont: forall s el1 el2 k,
  multi_estep s el1 el2 ->
  multi_cstep (CL_ECont el1 k, s) (CL_ECont el2 k, s).
Proof.
  intros.
  induction_1n H.
  + reflexivity.
  + transitivity_1n (CL_ECont el0 k, s).
    - apply CS_ECont; tauto.
    - tauto.
Qed.
```

```
Lemma MCS_CCont: forall cl1 s1 cl2 s2 k,
  multi_cstep (cl1, s1) (cl2, s2) ->
  multi_cstep (CL_CCont cl1 k, s1) (CL_CCont cl2 k, s2).
Proof.
  intros.
  induction_1n H.
  + reflexivity.
  + transitivity_1n (CL_CCont cl0 k, s0).
    - apply CS_CCont; tauto.
    - tauto.
Qed.
```