



软件理论基础与实践

SmallStep: Small-Step Operational Semantics

熊英飞
北京大学



大步法 vs 小步法

- 之前所学的语义被称为“大步法(Big-Step)”语义
 - 直接一步给出程序执行的结果
 - 因为定义简单且简化证明，也被称为“自然语义”
- 但无法展现程序中间的执行过程
 - 交互式程序（运行不终止）
 - 中间卡住的程序执行状态
 - 并行程序语义
- 小步法(Small-Step): 定义程序单步执行的语义



小型语言

语法

```
Inductive tm : Type :=  
  | C : nat -> tm          (* Constant *)  
  | P : tm -> tm -> tm.   (* Plus *)
```

大步法
语义

$$\frac{}{C\ n \Rightarrow n} \text{ (E_Const)}$$
$$\frac{t_1 \Rightarrow n_1 \quad t_2 \Rightarrow n_2}{P\ t_1\ t_2 \Rightarrow n_1 + n_2} \text{ (E_Plus)}$$



小型语言

大步法语义
定义为函数

```
Fixpoint evalF (t : tm) : nat :=  
  match t with  
  | C n => n  
  | P t1 t2 => evalF t1 + evalF t2  
  end.
```

大步法语义
定义为关系

```
Inductive eval : tm -> nat -> Prop :=  
  | E_Const : forall n,  
    C n ==> n  
  | E_Plus : forall t1 t2 n1 n2,  
    t1 ==> n1 ->  
    t2 ==> n2 ->  
    P t1 t2 ==> (n1 + n2)  
  
where " t '==>' n " := (eval t n).
```



小步法语义

$$\frac{}{P (C n_1) (C n_2) \rightarrow C (n_1 + n_2)} \quad (\text{ST_PlusConstConst})$$

$$\frac{t_1 \rightarrow t_1'}{P t_1 t_2 \rightarrow P t_1' t_2} \quad (\text{ST_Plus1})$$

$$\frac{t_2 \rightarrow t_2'}{P (C n_1) t_2 \rightarrow P (C n_1) t_2'} \quad (\text{ST_Plus2})$$

- 每次将一个可直接计算的子表达式换成常量
- 常量不用进一步计算
- 从左往右的顺序计算



小步法语义定义为关系

```
Inductive step : tm -> tm -> Prop :=  
  | ST_PlusConstConst : forall n1 n2,  
    P (C n1) (C n2) --> C (n1 + n2)  
  | ST_Plus1 : forall t1 t1' t2,  
    t1 --> t1' ->  
    P t1 t2 --> P t1' t2  
  | ST_Plus2 : forall n1 t2 t2',  
    t2 --> t2' ->  
    P (C n1) t2 --> P (C n1) t2'
```

where " t '-->' t' " := (step t t').



语义的性质：确定性

Definition `relation` (X : Type) := X -> X -> Prop.

Definition `deterministic` {X : Type} (R : relation X) :=
forall x y1 y2 : X, R x y1 -> R x y2 -> y1 = y2.

Theorem `step_deterministic`:
deterministic step.



确定性证明

Proof.

```
unfold deterministic. intros x y1 y2 Hy1 Hy2.  
(* Hy1: x --> y1  
   Hy2: x --> y2  
   Goal: y1 = y2 *)  
generalize dependent y2.  
induction Hy1; intros y2 Hy2.  
- (* ST_PlusConstConst *) inversion Hy2; subst.  
+ (* ST_PlusConstConst *) reflexivity.  
+ (* ST_Plus1 *) inversion H2.  
+ (* ST_Plus2 *) inversion H2.
```




确定性证明

```
- (* ST_Plus1 *) inversion Hy2; subst.  
  + (* ST_PlusConstConst *)  
    inversion Hy1.  
  + (* ST_Plus1 *)  
    rewrite <- (IHHy1 t1'0).  
    reflexivity. assumption.  
  + (* ST_Plus2 *)  
    inversion Hy1.  
- (* ST_Plus2 *) inversion Hy2; subst.  
  + (* ST_PlusConstConst *)  
    inversion Hy1.  
  + (* ST_Plus1 *) inversion H2.  
  + (* ST_Plus2 *)  
    rewrite <- (IHHy1 t2'0).  
    reflexivity. assumption.
```

Qed.



简化证明

- 证明需要反复调用inversion推出矛盾
- 定义策略在任意命题上反复应用n次inversion

```
Ltac solve_by_inverts n :=  
  match goal with | H : ?T |- _ =>  
  match type of T with Prop =>  
    solve [  
      inversion H;  
      match n with S (S (?n')) =>  
        subst; solve_by_inverts (S n')  
      end ]  
  end end.
```

```
Ltac solve_by_invert :=  
  solve_by_inverts 1.
```

|- : 匹配目标 (复习)
match type of X with
Type: 匹配类型
solve [策略]: 如果策略
没有完成当前目标证明,
就报错
(避免进入执行完策略
但没有证明目标的情况)



简化证明

Theorem `step_deterministic_alt`: deterministic step.

Proof.

```
intros x y1 y2 Hy1 Hy2.
generalize dependent y2.
induction Hy1; intros y2 Hy2;
  inversion Hy2; subst; try solve_by_invert.
- (* ST_PlusConstConst *) reflexivity.
- (* ST_Plus1 *)
  apply IHHy1 in H2. rewrite H2. reflexivity.
- (* ST_Plus2 *)
  apply IHHy1 in H2. rewrite H2. reflexivity.
```

Qed.



区分正常结束状态

- 如果当前状态不能再往下执行，如何知道已经是正常结束还是程序出现错误？

```
Inductive value : tm -> Prop :=  
  | v_const : forall n, value (C n).
```

- 用值来改写语义规则

$$\frac{}{P (C n_1) (C n_2) \rightarrow C (n_1 + n_2)} \text{ (ST_PlusConstConst)}$$

$$\frac{t_1 \rightarrow t_1'}{P t_1 t_2 \rightarrow P t_1' t_2} \text{ (ST_Plus1)}$$

$$\frac{\text{value } v_1 \quad t_2 \rightarrow t_2'}{P v_1 t_2 \rightarrow P v_1 t_2'} \text{ (ST_Plus2)}$$



改写Coq定义

Reserved Notation " t '-->' t' " (at level 40).

```
Inductive step : tm -> tm -> Prop :=
| ST_PlusConstConst : forall n1 n2,
    P (C n1) (C n2)
  --> C (n1 + n2)
| ST_Plus1 : forall t1 t1' t2,
    t1 --> t1' ->
    P t1 t2 --> P t1' t2
| ST_Plus2 : forall v1 t2 t2',
    value v1 ->
    t2 --> t2' ->
    P v1 t2 --> P v1 t2'
```

where " t '-->' t' " := (step t t').



Strong Progress

Theorem `strong_progress` : forall t,
value t \setminus (exists t', t --> t').

Proof.

induction t.

- (* C *) left. apply v_const.

- (* P *) right.

(* IHt1: value t1 \setminus (exists t' : tm, t1 --> t')
IHt2: value t2 \setminus (exists t' : tm, t2 --> t')
exists t' : tm, P t1 t2 --> t' *)

destruct IHt1 as [IHt1 | [t1' Ht1]].

+ (* l *) destruct IHt2 as [IHt2 | [t2' Ht2]].

* (* l *) inversion IHt1. inversion IHt2.

exists (C (n + n0)). apply ST_PlusConstConst.

* (* r *)

exists (P t1 t2'). apply ST_Plus2. apply IHt1. apply Ht2.

+ (* r *)

exists (P t1' t2). apply ST_Plus1. apply Ht1.

Qed.



标准型

- 定义不能约简的项为标准型
- Strong Progress告诉我们标准型是值
- 那么值一定是标准型吗?

```
Definition normal_form {X : Type}
  (R : relation X) (t : X) : Prop :=
  ~ exists t', R t t'.
```

```
Lemma value_is_nf : forall v, value v -> normal_form step v.
```

```
Lemma nf_is_value : forall t, normal_form step t -> value t.
```

```
Corollary nf_same_as_value : forall t,
  normal_form step t <-> value t.
```



多步约简关系

- 如何从小步法语义得到程序运行结果?
 - 首先定义多步约简关系
 - 然后将程序运行结果定义为多步约简关系可达的值

```
Inductive multi {X : Type} (R : relation X) : relation X :=
| multi_refl : forall (x : X), multi R x x
| multi_step : forall (x y z : X),
    R x y ->
    multi R y z ->
    multi R x z.
```

```
Notation " t '-->*' t' " := (multi step t t') (at level 40).
```




多步约简关系

- 多步约简是自反的，即包括0步约简
- 多步约简包含所有的单步约简

```
Theorem multi_R : forall (X : Type) (R : relation X) (x y : X),  
  R x y -> (multi R) x y.
```

- 多步约简是传递的

```
Theorem multi_trans :  
  forall (X : Type) (R : relation X) (x y z : X),  
    multi R x y ->  
    multi R y z ->  
    multi R x z.
```



标准化

- 非标准型是否一定会规约到标准型？

```
Definition normalizing {X : Type} (R : relation X) :=  
  forall t, exists t',  
    (multi R) t t' /\ normal_form R t'.
```

```
Theorem step_normalizing :  
  normalizing step.
```

- 证明思路：在tm的定义上归纳



大步法和小步法的等价性

`Theorem eval__multistep : forall t n,
t ==> n -> t -->* C n.`

- 证明思路：在 \Rightarrow 上做归纳
- 具体证明留作作业
- 课本提供了两个引理帮助证明

`Lemma multistep_congr_1 : forall t1 t1' t2,
t1 -->* t1' ->
P t1 t2 -->* P t1' t2.`

`Lemma multistep_congr_2 : forall t1 t2 t2',
value t1 ->
t2 -->* t2' ->
P t1 t2 -->* P t1 t2'.`



大步法和小步法的等价性

Definition `normal_form_of` (t t' : tm) :=
(t -->* t' /\ normal_form step t').

Theorem `multistep_eval` : forall t t',
normal_form_of t t' -> exists n, t' = C n /\ t ==> n.

- 证明思路：假设 $t' = C n$ ，然后对 $-->^*$ 做归纳
- 课本提供如下引理来辅助证明

Lemma `step_eval` : forall t t' n,
t --> t' ->
t' ==> n ->
t ==> n.



IMP的小步法语义: aexp

- 需要将状态加入约简关系

```
Inductive aval : aexp -> Prop :=  
  | av_num : forall n, aval (ANum n).
```

```
Reserved Notation " a '/' st '-->a' a' "  
  (at level 40, st at level 39).
```

```
Inductive astep (st : state) : aexp -> aexp -> Prop :=  
  | AS_Id : forall i,  
    AId i / st -->a ANum (st i)  
  | AS_Plus1 : forall a1 a1' a2,  
    a1 / st -->a a1' ->  
    (APlus a1 a2) / st -->a (APlus a1' a2)  
  | AS_Plus2 : forall v1 a2 a2',  
    aval v1 ->  
    a2 / st -->a a2' ->  
    (APlus v1 a2) / st -->a (APlus v1 a2')
```



IMP的小步法语义: aexp

```
| AS_Plus : forall n1 n2,  
  APlus (ANum n1) (ANum n2) / st -->a ANum (n1 + n2)  
| AS_Minus1 : forall a1 a1' a2,  
  a1 / st -->a a1' ->  
  (AMinus a1 a2) / st -->a (AMinus a1' a2)  
| AS_Minus2 : forall v1 a2 a2',  
  aval v1 ->  
  a2 / st -->a a2' ->  
  (AMinus v1 a2) / st -->a (AMinus v1 a2')  
| AS_Minus : forall n1 n2,  
  (AMinus (ANum n1) (ANum n2)) / st -->a (ANum (minus n1 n2))
```



IMP的小步法语义: aexp

```
| AS_Mult1 : forall a1 a1' a2,  
  a1 / st -->a a1' ->  
  (AMult a1 a2) / st -->a (AMult a1' a2)  
| AS_Mult2 : forall v1 a2 a2',  
  aval v1 ->  
  a2 / st -->a a2' ->  
  (AMult v1 a2) / st -->a (AMult v1 a2')  
| AS_Mult : forall n1 n2,  
  (AMult (ANum n1) (ANum n2)) / st -->a (ANum (mult n1 n2))  
  
where " a '/' st '-->a' a' " := (astep st a a').
```



IMP的小步法语义: bexp

Reserved Notation " b '/' st '-->b' b' "
(at level 40, st at level 39).

```
Inductive bstep (st : state) : bexp -> bexp -> Prop :=
| BS_Eq1 : forall a1 a1' a2,
  a1 / st -->a a1' ->
  (BEq a1 a2) / st -->b (BEq a1' a2)
| BS_Eq2 : forall v1 a2 a2',
  aval v1 ->
  a2 / st -->a a2' ->
  (BEq v1 a2) / st -->b (BEq v1 a2')
| BS_Eq : forall n1 n2,
  (BEq (ANum n1) (ANum n2)) / st -->b
  (if (n1 =? n2) then BTrue else BFalse)
```




IMP的小步法语义: bexp

```
| BS_LtEq1 : forall a1 a1' a2,  
  a1 / st -->a a1' ->  
  (BLe a1 a2) / st -->b (BLe a1' a2)  
| BS_LtEq2 : forall v1 a2 a2',  
  aval v1 ->  
  a2 / st -->a a2' ->  
  (BLe v1 a2) / st -->b (BLe v1 a2')  
| BS_LtEq : forall n1 n2,  
  (BLe (ANum n1) (ANum n2)) / st -->b  
  (if (n1 <=? n2) then BTrue else BFalse)  
| BS_NotStep : forall b1 b1',  
  b1 / st -->b b1' ->  
  (BNot b1) / st -->b (BNot b1')  
| BS_NotTrue : (BNot BTrue) / st -->b BFalse  
| BS_NotFalse : (BNot BFalse) / st -->b BTrue
```



IMP的小步法语义: bexp

```
| BS_AndStep : forall b1 b1' b2,  
    b1 / st -->b b1' ->  
    (BAnd b1 b2) / st -->b (BAnd b1' b2)  
| BS_AndTrueStep : forall b2 b2',  
    b2 / st -->b b2' ->  
    (BAnd BTrue b2) / st -->b (BAnd BTrue b2')  
| BS_AndFalse : forall b2,  
    (BAnd BFalse b2) / st -->b BFalse  
| BS_AndTrueTrue : (BAnd BTrue BTrue) / st -->b BTrue  
| BS_AndTrueFalse : (BAnd BTrue BFalse) / st -->b BFalse
```

where " b '/' st '-->b' b' " := (bstep st b b').



IMP的小步法语义: cmd

- 不同点之一: cmd会修改状态
 - 在约简关系的前后都加入状态
- 不同点之二: 表达式约简成常量值, 那么命令应该约简成什么?
 - 单个命令最后约简为skip, Sequence负责去掉skip



IMP的小步法语义: cmd

Reserved Notation " t '/' st '-->' t' '/' st' "
(at level 40, st at level 39, t' at level 39).

```
Inductive cstep : (com * state) -> (com * state) -> Prop :=
| CS_AssStep : forall st i a1 a1',
  a1 / st -->a a1' ->
  <{ i := a1 }> / st --> <{ i := a1' }> / st
| CS_Ass : forall st i n,
  <{ i := ANum n }> / st --> <{ skip }> / (i !-> n ; st)
| CS_SeqStep : forall st c1 c1' st' c2,
  c1 / st --> c1' / st' ->
  <{ c1 ; c2 }> / st --> <{ c1' ; c2 }> / st'
| CS_SeqFinish : forall st c2,
  <{ skip ; c2 }> / st --> c2 / st
```



IMP的小步法语义: cmd

```
| CS_IfStep : forall st b1 b1' c1 c2,  
  b1 / st --> b b1' ->  
  <{ if b1 then c1 else c2 end }> / st  
  -->  
  <{ if b1' then c1 else c2 end }> / st  
| CS_IfTrue : forall st c1 c2,  
  <{ if true then c1 else c2 end }> / st --> c1 / st  
| CS_IfFalse : forall st c1 c2,  
  <{ if false then c1 else c2 end }> / st --> c2 / st  
| CS_While : forall st b1 c1,  
  <{ while b1 do c1 end }> / st  
  -->  
  <{ if b1 then c1; while b1 do c1 end else skip end }> / st
```

where " t '/' st '-->' t' '/' st' " := (cstep (t,st) (t',st')).



定义并行IMP

```
Inductive com : Type :=
| CSkip : com
| CAss : string -> aexp -> com
| CSeq : com -> com -> com
| CIf : bexp -> com -> com -> com
| CWhile : bexp -> com -> com
| CPar : com -> com -> com.
```

```
Notation "'par' c1 'with' c2 'end'" :=
  (CPar c1 c2)
  (in custom com at level 0, c1 at level
  99, c2 at level 99).
```

为什么par的语义不容易用大步法定义?



定义并行IMP

Inductive `cstep` : (com * state) -> (com * state) -> **Prop** :=

```
.....
| CS_Par1 : forall st c1 c1' c2 st',
  c1 / st --> c1' / st' ->
  <{par c1 with c2 end}> / st --> <{par c1' with c2 end}> / st'
| CS_Par2 : forall st c1 c2 c2' st',
  c2 / st --> c2' / st' ->
  <{par c1 with c2 end}> / st --> <{par c1 with c2' end}> / st'
| CS_ParDone : forall st,
  <{par skip with skip end}> / st --> <{skip}> / st
```



论证并行程序的性质

Definition `cmultistep` := multi cstep.

Definition `par_loop` : com :=

```
<{  
  par  
    Y := 1  
  with  
    while (Y = 0) do  
      X := X + 1  
    end  
end}>.
```

Theorem `par_loop_any_X`:

```
forall n, exists st',  
  par_loop / empty_st -->* <{skip}> / st'  
  /\ st' X = n.
```




自动化小步法执行

```
Example step_example1 :  
  (P (C 3) (P (C 3) (C 4)))  
  -->* (C 10).
```

Proof.

```
  apply multi_step with (P (C 3) (C 7)).  
    apply ST_Plus2.  
      apply v_const.  
        apply ST_PlusConstConst.  
  apply multi_step with (C 10).  
    apply ST_PlusConstConst.  
  apply multi_refl.
```

Qed.

全是apply，可以用auto解决



自动化小步法执行

Hint Constructors step value multi: core.

```
Example step_example1' :  
  (P (C 3) (P (C 3) (C 4)))  
  -->* (C 10).
```

Proof.

```
eauto.
```

Qed.

但是，这样无法展现小步法的中间执行过程，也无法在不给出结果的情况下求值



自动化小步法执行

- 首先，不将multi加入auto可以搜索的Constructor范
- 证明的每一行和一步约简对应

```
Hint Constructors step value: core.
```

```
Example step_example1' :  
  (P (C 3) (P (C 3) (C 4)))  
  -->* (C 10).
```

```
Proof.
```

```
  eapply multi_step. auto. simpl.  
  eapply multi_step. auto. simpl.  
  apply multi_refl.
```

```
Qed.
```



自动化小步法执行

- 用策略来自动化该过程并打印每一步的结果

```
Tactic Notation "print_goal" :=  
  match goal with |- ?x => idtac x end.
```

```
Tactic Notation "normalize" :=  
  repeat (print_goal; eapply multi_step ;  
          [ (eauto 10; fail) | (instantiate; simpl)]);  
  apply multi_refl.
```

Idtac: 打印对应符号

Instantiate: 替换存在变量（在句号的时候自动做，但此处无句号，所以手动）

1. [XXX | XXX]是什么意思？
2. 为什么 eauto 10 之后要接 fail？



答案1

- 分号策略的标准型： $T; [T1 \mid T2 \mid \dots \mid Tn]$
 - 首先应用T，然后把T1..Tn分别应用到T生成的子目标上
- 应用 `eapply multi_step` 之后会生成两个目标
 - 原表达式 $--> ?y$
 - $?y -->^*$ 最终目标
- 第一个用 `eauto` 证明，第二个只需要 `simpl` 即可



答案2

- T1;T2: 将T2应用到T1所产生的的每一个Goal上
- (eauto 10; fail)在eauto完成证明的时候不会失败，否则会失败
 - fail在当前没有goal的时候不会执行
 - 和之前solve起的效果类似



采用normalize

```
Example step_example1'' :  
  (P (C 3) (P (C 3) (C 4)))  
  -->* (C 10).
```

Proof.

```
normalize.
```

```
(* The [print_goal] in the [normalize] tactic shows  
   a trace of how the expression reduced...
```

```
  (P (C 3) (P (C 3) (C 4))) -->* C 10)
```

```
  (P (C 3) (C 7)) -->* C 10)
```

```
  (C 10 -->* C 10)
```

```
*)
```

Qed.



采用normalize

- 甚至可以不提供最终结果就可以靠normalize直接约简到标准型

```
Example step_example1'' : exists e',  
  (P (C 3) (P (C 3) (C 4)))  
  -->* e'.
```

Proof.

```
eexists. normalize.
```

Qed.



作业

- 完成SmallStep中standard非optional并不属于Additional Exercises的8道习题
 - 请使用最新英文版教材
 - 如有时间，推荐完成par_body_n__Sn