软件理论基础与实践

Imp: Simple Imperative Programs

熊英飞
北京大学

课程进展

- Coq √
- 数理逻辑基础 √
- 形式语义
- 类型系统

高可信软件验证

- Coq要求递归为（Coq能分析出来的）结构递归
- 大量程序根本无法用Coq写出
- 如何采用Coq进行高可信软件验证?

- 首先在Coq中定义出一个程序设计语言
  - 该语言的程序是符合某种自定义类型的数据，不受Coq递归的限制
  - 需要定义语法
- 然后论证该语言对应程序的性质
  - 需要定义语义

本次课目标

- 定义一个命令式程序设计语言IMP

```
Z := X;
        Y := 1;
        while ~(Z = 0) do
          Y := Y × Z;
          Z := Z - 1
        end
```

- 先从定义语言的表达式开始

定义语法

- 语法定义通常采用上下文无关文法并写作BNF形式

```
a  :=  nat
    |  a + a
    |  a - a
    |  a × a

b  :=  true
    |  false
    |  a = a
    |  a ≠ a
    |  a ≤ a
    |  a > a
    |  ¬b
    |  b && b
```

定义语法

- 上下文无关文法记录为Coq的归纳定义

```
a  := nat
    | a + a
    | a − a
    | a × a

b  := true
    | false
    | a = a
    | a ≠ a
    | a ≤ a
    | a > a
    | ¬b
    | b && b
```

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).

Inductive bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : aexp)
  | BNeq (a1 a2 : aexp)
  | BLe (a1 a2 : aexp)
  | BGt (a1 a2 : aexp)
  | BNot (b : bexp)
  | BAnd (b1 b2 : bexp).
```

形式语义

- 操作语义
  - 将程序元素解释为计算步骤
- 指称语义
  - 将程序元素解释为数学中严格定义的对象，通常为函数
- 操作语义 vs 指称语义
  - 在现代程序语义学的教材中，二者通常是等价的，只是惯用符号不同
  - 操作语义常采用逻辑推导规则描述，指称语义常采用集合定义描述
  - 本书采用集合定义的方式描述了自称为操作语义的语义
- 公理语义
  - 将程序元素解释为前条件和后条件，并可用霍尔逻辑推导
  - 将在《Programming Language Foundations》部分介绍

语义建模成函数

```
Fixpoint aeval (a : aexp) : nat :=
  match a with
  | ANum n => n
  | APlus  a1 a2 => (aeval a1) + (aeval a2)
  | AMinus a1 a2 => (aeval a1) - (aeval a2)
  | AMult  a1 a2 => (aeval a1) * (aeval a2)
  end.

Fixpoint beval (b : bexp) : bool :=
  match b with
  | BTrue       => true
  | BFalse      => false
  | BEq a1 a2   => (aeval a1) =? (aeval a2)
  | BNeq a1 a2  => negb ((aeval a1) =? (aeval a2))
  | BLe a1 a2   => (aeval a1) <=? (aeval a2)
  | BGt a1 a2   => negb ((aeval a1) <=? (aeval a2))
  | BNot b1     => negb (beval b1)
  | BAnd b1 b2  => andb (beval b1) (beval b2)
  end.
```

优化是从程序到程序的映射

```
Fixpoint optimize_0plus (a:aexp) : aexp :=
  match a with
  | ANum n => ANum n
  | APlus (ANum 0) e2 => optimize_0plus e2
  | APlus  e1 e2 => APlus  (optimize_0plus e1) (optimize_0plus e2)
  | AMinus e1 e2 => AMinus (optimize_0plus e1) (optimize_0plus e2)
  | AMult  e1 e2 => AMult  (optimize_0plus e1) (optimize_0plus e2)
  end.
```

这种基于程序结构的翻译过程常被称为“语法制导的翻译”

优化的正确性

```
Theorem optimize_0plus_sound: forall a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a. induction a.
  - (* ANum *) reflexivity.
  - (* APlus *) destruct a1 eqn:Ea1.
    + (* a1 = ANum n *) destruct n eqn:En.
      * (* n = 0 *)  simpl. apply IHa2.
      * (* n <> 0 *) simpl. rewrite IHa2. reflexivity.
    + (* a1 = APlus a1_1 a1_2 *)
      simpl. simpl in IHa1. rewrite IHa1. rewrite IHa2. reflexivity.
    + (* a1 = AMinus a1_1 a1_2 *)
      simpl. simpl in IHa1. rewrite IHa1. rewrite IHa2. reflexivity.
    + (* a1 = AMult a1_1 a1_2 *)
      simpl. simpl in IHa1. rewrite IHa1. rewrite IHa2. reflexivity.
  - (* AMinus *)
    simpl. rewrite IHa1. rewrite IHa2. reflexivity.
  - (* AMult *)
    simpl. rewrite IHa1. rewrite IHa2. reflexivity.  Qed.
```

优化的正确性

```
Theorem optimize_0plus_sound: forall a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a. induction a.
  - (* ANum *) reflexivity.
  - (* APlus *) destruct a1 eqn:Ea1.
    + (* a1 = ANum n *) destruct n eqn:En.
      * (* n = 0 *)  simpl. apply IHa2.
      * (*
    + (* a
      simp                                                 reflexivity.
    + (* a
      simpl. simpl in IHa1. rewrite IHa1. rewrite IHa2. reflexivity.
    + (* a1 = AMult a1_1 a1_2 *)
      simpl. simpl in IHa1. rewrite IHa1. rewrite IHa2. reflexivity.
  - (* AMinus *)
    simpl. rewrite IHa1. rewrite IHa2. reflexivity.
  - (* AMult *)
    simpl. rewrite IHa1. rewrite IHa2. reflexivity.  Qed.
```

需要减少定理证明中的重复

try策略

```
Theorem silly1 : forall ae, aeval ae = aeval ae.
Proof. try reflexivity. (* This just does [reflexivity]. *) Qed.

Theorem silly2 : forall (P : Prop), P -> P.
Proof.
  intros P HP.
  try reflexivity. (* Just [reflexivity] would have failed. *)
  apply HP. (* We can still finish the proof in some other way. *)
Qed.
```

try T会应用T，如果T失败，就当什么都没发生

分号策略

T;T’首先应用T，然后将T’应用到所有T产生的子目标上。

```
Lemma foo : forall n, 0 <=? n = true.
Proof.
  intros.
  destruct n.
    - (* n=0 *) simpl. reflexivity.
    - (* n=S n' *) simpl. reflexivity.
Qed.
```

```
Lemma foo' : forall n, 0 <=? n = true.
Proof.
  intros.
  destruct n;
  simpl;
  reflexivity.
Qed.
```

用分号策略改写证明

```
Theorem optimize_0plus_sound'': forall a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;
    (* Most cases follow directly by the IH *)
    try (simpl; rewrite IHa1; rewrite IHa2; reflexivity);
    (* ... or are immediate by definition *)
    try reflexivity.
  (* The interesting case is when a = APlus a1 a2. *)
  - (* APlus *)
    destruct a1; try (simpl; simpl in IHa1; rewrite IHa1;
                      rewrite IHa2; reflexivity).
    + (* a1 = ANum n *) destruct n;
      simpl; rewrite IHa2; reflexivity. Qed.
```

分号策略的通用形式

- T; [T1 | T2 | … | Tn]
  - 首先应用T，然后把T1..Tn分别应用到T生成的子目标上
- T;T’是T; [T’ | T’ | … | T’]的简写形式

repeat策略

repeat T反复应用T直到失败

```
Theorem In10 : In 10 [1;2;3;4;5;6;7;8;9;10].
Proof.
  repeat (try (left; reflexivity); right).
Qed.
```

```
Theorem repeat_loop : forall (m n : nat),
    m + n = n + m.
Proof.
  intros m n.
  repeat rewrite Nat.add_comm.
  (* 死机 *)
```

定义简单的策略

- 用Tactic Notation可以定义简单的策略

```
Tactic Notation "invert" hyp(H) :=
  inversion H;
  subst;
  clear H.
```

- 更复杂的策略可以用Ltac定义

```
Ltac invert H := inversion H; subst; clear H.
```

- 后续课程中会涉及部分Ltac定义，但证明策略不是本课程内容
  - 详细学习《Certified Programming with dependent types》

lia策略

- 可用于快速证明线性表达式定理

```
Example silly_presburger_example : forall m n o p,
  m + n <= n + o /\ o + 3 = p + 3 ->
  m <= p.
Proof.
  intros. lia.
Qed.
```

```
Example plus_comm__omega : forall m n,
    m + n = n + m.
Proof.
  intros. lia.
Qed.
```

- 如果原命题无法被证明或证明失败（无法区分二者），则策略应用失败

其他一些策略

- clear H: 删除假设H
- subst x: 如果存在x=e或者e=x的假设，则删除该假设并把x替换成e
- subst: 对所有变量应用上述策略
- rename... into...: 对变量/假设改名
- assumption: 寻找和目标一样的假设并应用
- contradiction: 寻找和False等价的假设并推出矛盾
- constructor: 寻找一个可以匹配目标的归纳定义构造函数c，并执行apply c。

语义建模成函数的问题

- Coq中的函数必须是全函数

```
Inductive aexp : Type :=
   | ANum (n : nat)
   | APlus (a1 a2 : aexp)
   | AMinus (a1 a2 : aexp)
   | AMult (a1 a2 : aexp)
   | ADiv (a1 a2 : aexp).
```

```
Inductive aexp : Type :=
   | AAny    (* 产生随机数 *)
   | ANum (n : nat)
   | APlus (a1 a2 : aexp)
   | AMinus (a1 a2 : aexp)
   | AMult (a1 a2 : aexp).
```

- Coq中的函数必须在Coq编译器的分析能力内能终止
  - while(true);
  - while($n^5 + n = 34$) n++;

语义建模成关系

```
Inductive aevalR : aexp -> nat -> Prop :=
  | E_ANum (n : nat) :
      aevalR (ANum n) n
  | E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 ->
      aevalR e2 n2 ->
      aevalR (APlus e1 e2) (n1 + n2)
  | E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 ->
      aevalR e2 n2 ->
      aevalR (AMinus e1 e2) (n1 - n2)
  | E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
      aevalR e1 n1 ->
      aevalR e2 n2 ->
      aevalR (AMult e1 e2) (n1 * n2).
```

语义建模

先保留符号，就可以在定义过程中递归使用

```
Reserved Notation "e '==>' n" (at level 90, left associativity).

Inductive aevalR : aexp -> nat -> Prop :=
  | E_ANum (n : nat) :
      (ANum n) ==> n
  | E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) -> (e2 ==> n2) -> (APlus e1 e2) ==> (n1 + n2)
  | E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) -> (e2 ==> n2) -> (AMinus e1 e2) ==> (n1 - n2)
  | E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) -> (e2 ==> n2) -> (AMult e1 e2) ==> (n1 * n2)
  | E_ADiv (a1 a2 : aexp) (n1 n2 n3 : nat) :
      (a1 ==> n1) -> (a2 ==> n2) -> (n2 > 0) ->
      (mult n2 n3 = n1) -> (ADiv a1 a2) ==> n3

  where "e '==>' n" := (aevalR e n) : type_scope.
```

语义建模成关系（推导规则）

$$\frac{}{\text{ANum } n \Longrightarrow n}$$

$$\frac{e_1 \Longrightarrow n_1 \qquad e_2 \Longrightarrow n_2}{\text{APlus } e_1\ e_2 \Longrightarrow n_1 + n_2}$$

$$\frac{e_1 \Longrightarrow n_1 \qquad e_2 \Longrightarrow n_2}{\text{AMinus } e_1\ e_2 \Longrightarrow n_1 - n_2}$$

$$\frac{e_1 \Longrightarrow n_1 \qquad e_2 \Longrightarrow n_2}{\text{AMult } e_1\ e_2 \Longrightarrow n_1 * n_2}$$

$$\frac{e_1 \Longrightarrow n_1 \qquad e_2 \Longrightarrow n_2 \qquad n_2 * n_3 = n_1 \qquad n_2 > 0}{\text{ADiv } e_1\ e_2 \Longrightarrow n_3}$$

关系 vs 函数

- 建模成关系更自由，有时也更方便
  - 本身就不是函数
  - 较难在Coq中建模成函数
  - Coq自动生成的归纳定理会让一些证明更方便
- 建模成函数通常更方便
  - 蕴含了确定性和全函数的性质
  - 支持simpl
  - 函数还可以直接用在表达式中
- 大型Coq证明中也会两者都用，再另外证明定理说明两者等价

带变量的表达式

```
Definition state := total_map nat.

Inductive aexp : Type :=
   | ANum (n : nat)
   | AId (x : string)
   | APlus (a1 a2 : aexp)
   | AMinus (a1 a2 : aexp)
   | AMult (a1 a2 : aexp).
```

解析表达式

```
Definition W : string := "W".
Definition X : string := "X".
Definition Y : string := "Y".
Definition Z : string := "Z".

Coercion AId : string >-> aexp.
Coercion ANum : nat >-> aexp.

Declare Custom Entry com.
Declare Scope com_scope.
Notation "<{ e }>" := e
   (at level 0, e custom com at level 99) : com_scope.
```

声明隐式转换

创建专用语法com

<{ }>中的表达式用专用语法解析

解析表达式

```
Notation "( x )" := x (in custom com, x at level 99) : com_scope.
Notation "x" := x (in custom com at level 0, x constr at level 0) : com_scope.
Notation "f x .. y" := (.. (f x) .. y)
                  (in custom com at level 0, only parsing,
                  f constr at level 0, x constr at level 9,
                  y constr at level 9) : com_scope.
Notation "x + y" := (APlus x y) (in custom com at level 50, left associativity).
Notation "x - y" := (AMinus x y) (in custom com at level 50, left associativity).
Notation "x * y" := (AMult x y) (in custom com at level 40, left associativity).
Notation "'true'"  := true (at level 1).
Notation "'true'"  := BTrue (in custom com at level 0).
Notation "'false'"  := false (at level 1).
Notation "'false'"  := BFalse (in custom com at level 0).
Notation "x <= y" := (BLe x y) (in custom com at level 70, no associativity).
Notation "x > y"   := (BGt x y) (in custom com at level 70, no associativity).
Notation "x = y"  := (BEq x y) (in custom com at level 70, no associativity).
Notation "x <> y"  := (BNeq x y) (in custom com at level 70, no associativity).
Notation "x && y" := (BAnd x y) (in custom com at level 80, left associativity).
Notation "'~' b"  := (BNot b) (in custom com at level 75, right associativity).

Open Scope com_scope.
```

解析表达式

```
Definition example_aexp : aexp := <{ 3 + (X * 2) }>.
Definition example_bexp : bexp := <{ true && ~(X <= 4) }>.
```

修改语义函数

```
Fixpoint aeval (st : state) (a : aexp) : nat :=
  match a with
  | ANum n => n
  | AId x => st x  (* <--- 新增 *)
  | <{a1 + a2}> => (aeval st a1) + (aeval st a2)
  | <{a1 - a2}> => (aeval st a1) - (aeval st a2)
  | <{a1 * a2}> => (aeval st a1) * (aeval st a2)
  end.

Fixpoint beval (st : state) (b : bexp) : bool :=
  match b with
  | <{true}>      => true
  | <{false}>     => false
  | <{a1 = a2}>   => (aeval st a1) =? (aeval st a2)
  | <{a1 <> a2}>  => negb ((aeval st a1) =? (aeval st a2))
  | <{a1 <= a2}>  => (aeval st a1) <=? (aeval st a2)
  | <{a1 > a2}>   => negb ((aeval st a1) <=? (aeval st a2))
  | <{~ b1}>      => negb (beval st b1)
  | <{b1 && b2}>  => andb (beval st b1) (beval st b2)
  end.
```

程序示例

```
Definition empty_st := (_ !-> 0).

Notation "x '!->' v" := (t_update empty_st x v) (at level 100).

Example aexp1 :
    aeval (X !-> 5) <{ (3 + (X * 2))}>
  = 13.
Proof. reflexivity. Qed.

Example bexp1 :
    beval (X !-> 5) <{ true && ~(X <= 4)}>
  = true.
Proof. reflexivity. Qed.
```

IMP顶层语法

```
c := skip
   | x := a
   | c ; c
   | if b then c else c end
   | while b do c end
```

```
Inductive com : Type :=
  | CSkip
  | CAss (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).
```

解析程序

```
Notation "'skip'"  :=
         CSkip (in custom com at level 0) : com_scope.
Notation "x := y"  :=
         (CAss x y)
            (in custom com at level 0, x constr at level 0,
             y at level 85, no associativity) : com_scope.
Notation "x ; y" :=
         (CSeq x y)
           (in custom com at level 90, right associativity) : com_scope.
Notation "'if' x 'then' y 'else' z 'end'" :=
         (CIf x y z)
           (in custom com at level 89, x at level 99,
            y at level 99, z at level 99) : com_scope.
Notation "'while' x 'do' y 'end'" :=
         (CWhile x y)
            (in custom com at level 89, x at level 99,
             y at level 99) : com_scope.
```

程序示例

```
Definition fact_in_coq : com :=
  <{ Z := X;
     Y := 1;
     while ~(Z = 0) do
       Y := Y * Z;
       Z := Z - 1
     end }>.

Print fact_in_coq.
(* fact_in_coq =
<{ Z := X; Y := 1; while ~ Z = 0 do Y := Y * Z; Z := Z - 1 end }>
   : com *)
```

控制打印内容

```
Unset Printing Notations.
Print fact_in_coq.
(* ===>
   fact_in_coq =
   CSeq (CAss Z X)
        (CSeq (CAss Y (S O))
              (CWhile (BNot (BEq Z O))
                      (CSeq (CAss Y (AMult Y Z))
                            (CAss Z (AMinus Z (S O))))))
        : com *)
Set Printing Notations.
```

控制打印内容

```
Set Printing Coercions.
Print fact_in_coq.
(* ===>
   fact_in_coq =
   <{ Z := (AId X);
      Y := (ANum 1);
      while ~ (AId Z) = (ANum 0) do
        Y := (AId Y) * (AId Z);
        Z := (AId Z) - (ANum 1)
      end }>
        : com *)
Unset Printing Coercions.
```

Locate命令

- 之前学过Search命令用于查找符合条件的定义或定理，输出名字和类型
- Locate命令用于给出一个符号的完整名称或者Notation定义

```
Locate "&&".
(* ===>
    Notation
      "x && y" := BAnd x y (default interpretation)
      "x && y" := andb x y : bool_scope (default interpretation)
*)
```

Locate命令

```
Locate "while".
(* ===>
    Notation
      "'while' x 'do' y 'end'" :=
          CWhile x y : com_scope (default interpretation)
      "'_' '!->' v" := t_empty v (default interpretation)
*)
```

Locate命令

```
Locate aexp.
(* ===>
     Inductive LF.Imp.aexp
     Inductive LF.Imp.AExp.aexp
       (shorter name to refer to it in current context
        is AExp.aexp)
     Inductive LF.Imp.aevalR_division.aexp
       (shorter name to refer to it in current context
        is aevalR_division.aexp)
     Inductive LF.Imp.aevalR_extended.aexp
       (shorter name to refer to it in current context
        is aevalR_extended.aexp)
*)
```

命令的语义

$$\frac{}{\texttt{st =[skip]=> st}} \quad \text{(E\_Skip)}$$

$$\frac{\texttt{aeval st a = n}}{\texttt{st =[x := a]=> (x !-> n ; st)}} \quad \text{(E\_Ass)}$$

$$\frac{\begin{array}{c}\texttt{st =[}c_1\texttt{]=> st'}\\ \texttt{st' =[}c_2\texttt{]=> st''}\end{array}}{\texttt{st =[}c_1\texttt{;}c_2\texttt{]=> st''}} \quad \text{(E\_Seq)}$$

$$\frac{\begin{array}{c}\texttt{beval st b = true}\\ \texttt{st =[}c_1\texttt{]=> st'}\end{array}}{\texttt{st =[if b then }c_1\texttt{ else }c_2\texttt{ end]=> st'}} \quad \text{(E\_IfTrue)}$$

命令的语义

$$\frac{\texttt{beval st b = false} \qquad \texttt{st =[} c_2 \texttt{]=> st'}}{\texttt{st =[if b then } c_1 \texttt{ else } c_2 \texttt{ end]=> st'}} \quad \text{(E\_IfFalse)}$$

$$\frac{\texttt{beval st b = false}}{\texttt{st =[while b do c end]=> st}} \quad \text{(E\_WhileFalse)}$$

$$\frac{\texttt{beval st b = true} \qquad \texttt{st =[c]=> st'} \qquad \texttt{st' =[while b do c end]=> st''}}{\texttt{st =[while b do c end]=> st''}} \quad \text{(E\_WhileTrue)}$$

对应Coq关系

```
Reserved Notation
         "st '=[' c ']=>' st'"
         (at level 40, c custom com at level 99,
          st constr, st' constr at next level).

Inductive ceval : com -> state -> state -> Prop :=
  | E_Skip : forall st,
      st =[ skip ]=> st
  | E_Ass  : forall st a n x,
      aeval st a = n ->
      st =[ x := a ]=> (x !-> n ; st)
  | E_Seq : forall c1 c2 st st' st'',
      st  =[ c1 ]=> st'  ->
      st' =[ c2 ]=> st'' ->
      st  =[ c1 ; c2 ]=> st‘’
```

对应Coq关系

```
| E_IfTrue : forall st st' b c1 c2,
    beval st b = true ->
    st =[ c1 ]=> st' ->
    st =[ if b then c1 else c2 end]=> st'
| E_IfFalse : forall st st' b c1 c2,
    beval st b = false ->
    st =[ c2 ]=> st' ->
    st =[ if b then c1 else c2 end]=> st'
| E_WhileFalse : forall b st c,
    beval st b = false ->
    st =[ while b do c end ]=> st
| E_WhileTrue : forall st st' st'' b c,
    beval st b = true ->
    st  =[ c ]=> st' ->
    st' =[ while b do c end ]=> st'' ->
    st  =[ while b do c end ]=> st''

where "st =[ c ]=> st'" := (ceval c st st’).
```

程序运行举例

```
Example ceval_example1:
  empty_st =[
     X := 2;
     if (X <= 1)
       then Y := 3
       else Z := 4
     end
  ]=> (Z !-> 4 ; X !-> 2).
```

```
Proof.
  apply E_Seq with (X !-> 2).
  - (* assignment command *)
    apply E_Ass. reflexivity.
  - (* if command *)
    apply E_IfFalse.
    reflexivity.
    apply E_Ass. reflexivity.
Qed.
```

定理证明举例

```
Definition plus2 : com :=
  <{ X := X + 2 }>.

Theorem plus2_spec : forall st n st',
  st X = n ->
  st =[ plus2 ]=> st' ->
  st' X = n + 2.
```

```
Lemma t_update_eq : forall (A : Type) (m : total_map A) x v,
    (x !-> v ; m) x = v.
```

证明会用到t_update_eq，是在Maps一章证明的引理

定理证明举例

```
Proof.
  intros st n st' HX Heval.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   st : string -> nat
 *   n : nat
 *   st' : state
 *   HX : st X = n
 *   Heval : st =[ plus2 ]=> st'
 *   ============================
 *   st' X = n + 2
 *)
```

定理证明举例

```
 inversion Heval.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   st : string -> nat
 *   n : nat
 *   st' : state
 *   HX : st X = n
 *   Heval : st =[ plus2 ]=> st'
 *   st0 : state
 *   a : aexp
 *   n0 : nat
 *   x : string
 *   H3 : aeval st <{ X + 2 }> = n0
 *   H : x = X
 *   H1 : a = <{ X + 2 }>
 *   H0 : st0 = st
 *   H2 : (X !-> n0; st) = st'
 *   ==============================
 *   (X !-> n0; st) X = n + 2
 *)
```

定理证明举例

```
 subst.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   st : string -> nat
 *   Heval : st =[ plus2 ]=> (X !-> aeval st <{ X + 2 }>; st)
 *   ============================================================
 *   (X !-> aeval st <{ X + 2 }>; st) X = st X + 2
 *)
```

定理证明举例

```
 clear Heval.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   st : string -> nat
 *   =============================================
 *   (X !-> aeval st <{ X + 2 }>; st) X = st X + 2
 *)
```

定理证明举例

```
 simpl.
(** [Coq Proof View]
 * 1 subgoal
 *
 *   st : string -> nat
 *   ================================
 *   (X !-> st X + 2; st) X = st X + 2
 *)
  apply t_update_eq.
(** No more subgoals. *)
  Qed.
```

作业

- 完成Imp中standard非optional并不属于Additional Exercises的6道习题
  - 请使用最新英文版教材
  - 从本章起证明长度有显著增加，请尽早动手