软件理论基础与实践

Hoare2: Hoare Logic, Part II

熊英飞
北京大学

复习

- 霍尔逻辑的6条规则
- Assertion的定义
- 霍尔三元组的定义
- 解释如下式子的含义
  - X = 1 ->> Y = 1
  - forall st, <{X=1}> st -> <{Y=1}> st
  - forall st, X st = 1 st -> Y st = 1 st
  - forall st, st X = 1 -> st Y = 1

装饰程序Decorated Program

- 霍尔逻辑证明程序性质的过程基本和程序结构一致
- 可以用一种更紧凑的方式表达证明过程

```
  {{ X <= 3 }}
while X <= 2 do
  X := X + 1
end
  {{ X = 3 }}
```

装饰程序Decorated Program

```
  {{ X <= 3 }}
while X <= 2 do
    {{ X <= 3 /\ X <= 2 }} ->>
    {{ X + 1 <= 3 }}
  X := X + 1
    {{ X <= 3 }}
end
  {{ X <= 3 /\ ~(X <= 2) }} ->>
  {{ X = 3 }}
```

装饰程序与霍尔逻辑规则

```
{{ P }} skip {{ P }}
```

```
{{ P }} c₁; {{ Q }} c₂ {{ R }}
```

```
{{ P [X ⊢> a] }}
X := a
{{ P }}
```

```
{{ P }}
while b do
   {{ P ∧ b }}
   c₁
   {{ P }}
end
{{ P ∧ ¬b }}
```

```
{{ P }} ->> {{ P' }}
```

```
{{ P }}
if b then
   {{ P ∧ b }}
   c₁
   {{ Q }}
else
   {{ P ∧ ¬b }}
   c₂
   {{ Q }}
end
{{ Q }}
```

只需要检查每一部分的正确性就得到了整体证明的正确性

程序证明过程：顺序

```
(1) {{ X = m ∧ Y = n }}
        ->>
(2) {{ (X + Y) - ((X + Y) - Y) = n ∧ (X + Y) - Y = m }}
        X := X + Y;
(3) {{ X - (X - Y) = n ∧ X - Y = m }}
        Y := X - Y;
(4) {{ X - Y = n ∧ Y = m }}
        X := X - Y
(5) {{ X = n ∧ Y = m }}
```

程序证明过程：选择

```
(1)  {{True}}
       if X ≤ Y then
(2)  {{True ∧ X ≤ Y}}
     ->>
(3)  {{(Y - X) + X = Y ∨ (Y - X) + Y = X}}
         Z := Y - X
(4)  {{Z + X = Y ∨ Z + Y = X}}
       else
(5)  {{True ∧ ~(X ≤ Y) }}
     ->>
(6)  {{(X - Y) + X = Y ∨ (X - Y) + Y = X}}
         Z := X - Y
(7)  {{Z + X = Y ∨ Z + Y = X}}
       end
(8)  {{Z + X = Y ∨ Z + Y = X}}
```

程序证明过程：循环

```
(1) {{ True }}
      while ~(X = 0) do
(2) {{ True ∧ X ≠ 0 }}
    ->>
(3) {{ True }}
        X := X - 1
(4) {{ True }}
      end
(5) {{ True ∧ ~(X ≠ 0) }}
    ->>
(6) {{ X = 0 }}
```

从后条件获取循环不变式

```
(1) {{ True }}
     ->>
(2) {{ n × 0 + m = m }}
     X := m;
(3) {{ n × 0 + X = m }}
     Y := 0;
(4) {{ n × Y + X = m }}
     while n ≤ X do
(5) {{ n × Y + X = m ∧ n ≤ X }}
    ->>
(6) {{ n × (Y + 1) + (X - n) = m }}
       X := X - n;
(7) {{ n × (Y + 1) + X = m }}
       Y := Y + 1
(8) {{ n × Y + X = m }}
     end
(9) {{ n × Y + X = m ∧ ¬(n ≤ X) }}
    ->>
(10) {{ n × Y + X = m ∧ X < n }}
```

循环不变式的条件

- 足够弱：能被前条件推出
- 足够强：能推出后条件
- 能保持：每一次循环都保持条件

根据终止条件泛化

```
  {{ X = m ∧ Y = n }}
while ~(X = 0) do
  Y := Y - 1;
  X := X - 1
end
  {{ Y = n - m }}
```

- True作为循环不变式
    - 太弱，推不出后条件
- 后条件作为循环不变式
    - 太强，前条件推不出来，且循环也不保持
- 寻找一个条件，在X=0的时候等价于后条件
    - Y-X=n-m

根据终止条件泛化

```
(1)  {{ X = m ∧ Y = n }} ->> (a - OK)
(2)  {{ Y - X = n - m }}
        while ~(X = 0) do
(3)  {{ Y - X = n - m ∧ X ≠ 0 }} ->> (c - OK)
(4)  {{ (Y - 1) - (X - 1) = n - m }}
          Y := Y - 1;
(5)  {{ Y - (X - 1) = n - m }}
          X := X - 1
(6)  {{ Y - X = n - m }}
        end
(7)  {{ Y - X = n - m ∧ ~(X ≠ 0) }} ->> (b - OK)
(8)  {{ Y = n - m }}
```

练习：
为下面的证明找到循环不变式

```
{{ X = m }}
  while 2 ≤ X do
    X := X - 2
  end
{{ X = parity m }}
```

```
Fixpoint parity x :=
  match x with
  | 0 => 0
  | 1 => 1
  | S (S x') => parity x'
  end.
```

答案：根据终止条件泛化

```
{{ X = m }} ->>                                                 (a - OK)
{{ parity X = parity m }}
   while 2 ≤ X do
       {{ parity X = parity m ∧ 2 ≤ X }} ->>                    (c - OK)
       {{ parity (X-2) = parity m }}
         X := X - 2
       {{ parity X = parity m }}
   end
{{ parity X = parity m ∧ ~(2 ≤ X) }} ->>                        (b - OK)
{{ X = parity m }}
```

练习:
为下面的证明找到循环不变式

```
   {{ X=m }}
Z := 0;
while (Z+1)*(Z+1) ≤ X do
   Z := Z+1
end
   {{ Z×Z≤m ∧ m<(Z+1)*(Z+1) }}
```

答案：结合前后条件

```
    {{ X=m }} ->>
    {{ X=m ∧ 0*0 ≤ m }}
  Z := 0;
    {{ X=m ∧ Z×Z ≤ m }}
  while (Z+1)*(Z+1) ≤ X do
      {{ X=m ∧ Z×Z≤m ∧ (Z+1)*(Z+1)<=X }} ->>
      {{ X=m ∧ (Z+1)*(Z+1)<=m }}
    Z := Z + 1
      {{ X=m ∧ Z×Z≤m }}
  end
    {{ X=m ∧ Z×Z≤m ∧ ~((Z+1)*(Z+1)<=X) }} ->>
    {{ Z×Z≤m ∧ m<(Z+1)*(Z+1) }}
```

练习：为下面的证明找到循环不变式

```
   {{ X = m }}
Y := 0;
Z := 0;
while ~(Y = X) do
   Z := Z + X;
   Y := Y + 1
end
   {{ Z = m×m }}
```

答案：结合以上两种方法

```
    {{ X = m }} ->> (a - OK)
    {{ 0 = 0*m ∧ X = m }}
Y := 0;
    {{ 0 = Y×m ∧ X = m }}
Z := 0;
    {{ Z = Y×m ∧ X = m }}
while ~(Y = X) do
      {{ Z = Y×m ∧ X = m ∧ Y ≠ X }} ->> (c - OK)
      {{ Z+X = (Y+1)*m ∧ X = m }}
   Z := Z + X;
      {{ Z = (Y+1)*m ∧ X = m }}
   Y := Y + 1
      {{ Z = Y×m ∧ X = m }}
end
    {{ Z = Y×m ∧ X = m ∧ ~(Y ≠ X) }} ->> (b - OK)
    {{ Z = m×m }}
```

转换装饰程序到Coq证明

```
(1)  {{ True }}
        while ~(X = 0) do
(2)  {{ True ∧ X ≠ 0 }}
     ->>
(3)  {{ True }}
          X := X - 1
(4)  {{ True }}
        end
(5)  {{ True ∧ ~(X ≠ 0)  }}
     ->>
(6)  {{ X = 0 }}
```

转换装饰程序到Coq证明

```
Theorem reduce_to_zero_correct'' :
  {{True}}
  reduce_to_zero'
  {{X = 0}}.
Proof.
  unfold reduce_to_zero'.
  eapply hoare_consequence_post.
  - apply hoare_while.
    + eapply hoare_consequence_pre.
      * apply hoare_asgn.
      * assn_auto''.
  - (* fun st => True st /\ ~ (<{ ~X=0}> st)) ->> X = 0*)
  assn_auto''. (* doesn't succeed *)
Abort.
```

新的自动证明策略

```
Ltac verify_assn :=
  repeat split;
  simpl; unfold assert_implies;
  unfold ap in *; unfold ap2 in *;
  unfold bassn in *; unfold beval in *; unfold aeval in *;
  unfold assn_sub; intros;
  repeat (simpl in *;
          rewrite t_update_eq ||
          (try rewrite t_update_neq;
           [| (intro X; inversion X; fail)]));
  simpl in *;
  repeat match goal with [H : _ /\ _ |- _] => destruct H end;
  repeat rewrite not_true_iff_false in *;
  ……
```

无需了解细节，但对于大多数assign变换后的条件蕴含证明都可用

转换装饰程序到Coq证明

```
Theorem reduce_to_zero_correct''' :
  {{True}}
  reduce_to_zero'
  {{X = 0}}.
Proof.
  unfold reduce_to_zero'.
  eapply hoare_consequence_post.
  - apply hoare_while.
    + eapply hoare_consequence_pre.
      * apply hoare_asgn.
      * verify_assn.
  - verify_assn.
Qed.
```

能否自动化上述过程

- 定义装饰程序的语法，使得在Coq中可以直接书写
- 定义函数将装饰程序转化为命题
- 定义策略自动证明命题

装饰程序语法

避免重复，每种dcom默认只包括后条件，由decorated提供整个程序的前条件。

```
Inductive dcom : Type :=
| DCSkip (Q : Assertion)
  (* skip {{ Q }} *)
| DCSeq (d1 d2 : dcom)
  (* d1 ;; d2 *)
| DCAsgn (X : string) (a : aexp) (Q : Assertion)
  (* X := a {{ Q }} *)
| DCIf (b : bexp) (P1 : Assertion) (d1 : dcom)
       (P2 : Assertion) (d2 : dcom) (Q : Assertion)
  (* if b then {{ P1 }} d1 else {{ P2 }} d2 end {{ Q }} *)
| DCWhile (b : bexp) (P : Assertion) (d : dcom) (Q : Assertion)
  (* while b do {{ P }} d end {{ Q }} *)
| DCPre (P : Assertion) (d : dcom)
  (* ->> {{ P }} d *)
| DCPost (d : dcom) (Q : Assertion)
  (* d ->> {{ Q }} *).
Inductive decorated : Type :=
  | Decorated : Assertion -> dcom -> decorated.
```

装饰程序语法

```
Declare Scope dcom_scope.
Notation "'skip' {{ P }}"
      := (DCSkip P)
      (in custom com at level 0, P constr) : dcom_scope.
Notation "'while' b 'do' {{ Pbody }} d 'end' {{ Ppost }}"
      := (DCWhile b Pbody d Ppost)
           (in custom com at level 89, b custom com at level 99,
           Pbody constr, Ppost constr) : dcom_scope.
Notation "'if' b 'then' {{ P }} d 'else' {{ P' }} d' 'end' {{ Q }}"
      := (DCIf b P d P' d' Q)
           (in custom com at level 89, b custom com at level 99,
               P constr, P' constr, Q constr) : dcom_scope.
……
```

装饰程序书写实例

```
Example dec_while : decorated :=
  <{
  {{ True }}
  while ~(X = 0)
  do
    {{ True /\ (X <> 0) }}
    X := X - 1
    {{ True }}
  end
  {{ True /\  X = 0}} ->>
  {{ X = 0 }} }>.
```

从装饰程序变回普通程序

```
Fixpoint extract (d : dcom) : com :=
  match d with
  | DCSkip _             => CSkip
  | DCSeq d1 d2          => CSeq (extract d1) (extract d2)
  | DCAsgn X a _         => CAss X a
  | DCIf b _ d1 _ d2 _   => CIf b (extract d1) (extract d2)
  | DCWhile b _ d _      => CWhile b (extract d)
  | DCPre _ d            => extract d
  | DCPost d _           => extract d
  end.

Definition extract_dec (dec : decorated) : com :=
  match dec with
  | Decorated P d => extract d
  end.
```

获取装饰程序的前后条件

```
Definition pre_dec (dec : decorated) : Assertion :=
  match dec with
  | Decorated P d => P
  end.

Definition post_dec (dec : decorated) : Assertion :=
  match dec with
  | Decorated P d => post d
  end.
```

从装饰程序到命题

```
Fixpoint verification_conditions
         (P : Assertion) (d : dcom) : Prop :=
  match d with
  | DCSkip Q =>
      (P ->> Q)
  | DCSeq d1 d2 =>
      verification_conditions P d1
      /\ verification_conditions (post d1) d2
  | DCAsgn X a Q =>
      (P ->> Q [X |-> a])
  | DCIf b P1 d1 P2 d2 Q =>
      ((P /\ b) ->> P1)%assertion
      /\ ((P /\ ~ b) ->> P2)%assertion
      /\ (post d1 ->> Q) /\ (post d2 ->> Q)
      /\ verification_conditions P1 d1
      /\ verification_conditions P2 d2
```

从装饰程序到命题

```
| DCWhile b Pbody d Ppost =>
    (* post d is the loop invariant and the initial
       precondition *)
    (P ->> post d)
    /\ ((post d  /\ b) ->> Pbody)%assertion
    /\ ((post d  /\ ~ b) ->> Ppost)%assertion
    /\ verification_conditions Pbody d
 | DCPre P' d =>
    (P ->> P') /\ verification_conditions P' d
 | DCPost d Q =>
    verification_conditions P d /\ (post d ->> Q)
 end.
```

从装饰程序到命题：正确性

```
Theorem verification_correct : forall d P,
  verification_conditions P d -> {{P}} extract d {{post d}}.
Proof.
  induction d; intros; simpl in *.
  - (* Skip *)
    eapply hoare_consequence_pre.
      + apply hoare_skip.
      + assumption.
  (* 其他证明类似，略 *)
```

从装饰程序到命题：正确性

```
Definition dec_correct (dec : decorated) :=
  {{pre_dec dec}} extract_dec dec {{post_dec dec}}.

Definition verification_conditions_dec
           (dec : decorated) : Prop :=
  match dec with
  | Decorated P d => verification_conditions P d
  end.

Corollary verification_correct_dec : forall dec,
  verification_conditions_dec dec -> dec_correct dec.
```

自动证明

- 多数情况借助之前定义的verify_assn可自动证明

```
Ltac verify :=
  intros;
  apply verification_correct;
  verify_assn.

Theorem Dec_while_correct :
  dec_correct dec_while.
Proof. verify. Qed.
```

- 通常可以先尝试verify，对于证明不了的分支再手动证明

谓词转换计算

- 最弱前条件：{P}是c{Q}的最弱前条件，如果
  - {P}c{Q}
  - $\forall \mathrm{P}'.\{\mathrm{P}'\}\mathrm{c}\{\mathrm{Q}\} \Rightarrow P' \to P$
- 最强后条件：{Q}是{P}c的最强前条件，如果
  - {P}c{Q}
  - $\forall \mathrm{Q}'.\{\mathrm{P}\}\mathrm{c}\{\mathrm{Q}'\} \Rightarrow \mathrm{Q} \to \mathrm{Q}'$

- 最弱前条件计算：给定后条件和语句，求能形成霍尔三元组的最弱前条件
- 最强后条件计算：给定前条件和语句，求能形成霍尔三元组的最强后条件

最弱前条件计算

- $wp(skip, Q) = Q$

$$\textsc{Skip}\ \frac{}{\{P\}\ \textbf{skip}\ \{P\}}$$

- $wp(x \coloneqq a, Q) = Q[a/x]$

$$\textsc{Assign}\ \frac{}{\{P[a/x]\}\ x := a\ \{P\}}$$

- $wp(c_1; c_2, Q) = wp\big(c_1, wp(c_2, Q)\big)$

$$\textsc{Seq}\ \frac{\{P\}\ c_1\ \{R\} \qquad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1; c_2\ \{Q\}}$$

- $wp(if\ b\ then\ c_1\ else\ c_2, Q) = \big(b \rightarrow wp(c_1, Q)\big) \land \big(\neg b \rightarrow wp(c_2, Q)\big)$

$$\textsc{If}\ \frac{\{P \land b\}\ c_1\ \{Q\} \qquad \{P \land \neg b\}\ c_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

最弱前条件：举例

- wp(if (x > 0) x += 10; else x = 20, x>0)
  - =(x>0->wp(x+=10, x>0)) /\ (x<=0 -> wp(x=20, x>0))
  - =(x>0->x+10>0) /\ (x<=0 -> 20>0)
  - =True

最弱前条件计算：循环

- $wp(while\ b\ do\ c, Q) = \exists i \in Nat. L_i(Q)$
  - where
    - $L_0(Q) = false$
    - $L_{i+1}(Q) = (\neg b \Rightarrow Q) \wedge \left(b \Rightarrow wp\left(c, L_i(Q)\right)\right)$
- i代表循环最多执行了$i-1$次
- 注意这个最弱前条件蕴含了循环必然终止
  - 性质：$wp(c, false) = false$

$$\text{WHILE}\ \frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{P \wedge \neg b\}}$$

最强后条件计算

- $sp(P, skip) = P$
- $sp(P, x \coloneqq a) = \exists n.\ x = a[n/x] \land P[n/x]$
- $sp(P, c_1; c_2) = sp(sp(P, c_1), c_2)$
- $sp(P, if\ b\ then\ c_1\ else\ c_2) = sp(b \land P, c_1) \lor sp(\neg b \land P, c_2)$
- $sp(P, while\ b\ do\ c) = \neg b \land \exists i. L_i(P)$
  - where
    - $L_0(P) = P$
    - $L_{i+1}(P) = sp(b \land L_i(P), c)$

因为约束更复杂，实际使用较少

作业

- 完成Hoare2中standard非optional的4道习题
  - 请使用最新英文版教材