第二次习题课

王治奕

Loop_never_stops

```
Theorem loop_never_stops : ∀ st st',
  ~(st =[ loop ]⇒ st').
Proof.
  intros st st' contra. unfold loop in contra.
  remember <{ while true do skip end }> as loopdef
           eqn:Heqloopdef.
```

Loop_never_stops

```
induction contra; try discriminate.
```

```
st, st' : state
loopdef : com
Heqloopdef : loopdef = <{ while true do skip end }>
contra : st =[ loopdef ]⇒ st'
──────────────────────────────
⊥
```

Loop_never_stops

```
- inversion Heqloopdef; subst; discriminate.
```

```
b : bexp
c : com
Heqloopdef : <{ while b do c end }> = <{ while true do skip end }>
st : state
H : beval st b = false
______________________________________
⊥
```

Loop_never_stops

```
- apply IHcontra2; assumption.
```

```
b : bexp
c : com
Heqloopdef : <{ while b do c end }> = <{ while true do skip end }>
st, st', st'' : state
H : beval st b = true
contra1 : st =[ c ]⇒ st'
contra2 : st' =[ while b do c end ]⇒ st''
IHcontra1 : c = <{ while true do skip end }> → ⊥
IHcontra2 : <{ while b do c end }> = <{ while true do skip end }> → ⊥
______________________________________
⊥
```

Hoare_repeat : semantics

```
| E_RepeatTrue : ∀ b st st' c,
    st =[ c ]⇒ st' →
    beval st' b = true →
    st =[ repeat c until b end ]⇒ st'
| E_RepeatFalse : ∀ b st st' st'' c,
    st =[ c ]⇒ st' →
    beval st' b = false →
    st' =[ repeat c until b end ]⇒ st'' →
    st =[ repeat c until b end ]⇒ st''
```

Hoare_repeat : failed proof

```
Lemma hoare_repeat :
  ∀ P Q (b : bexp) c,
    {{ P }} c {{ Q }} →
    {{ Q ∧ ~b }} c {{ Q }} →
    {{ P }} repeat c until b end {{ Q ∧ b }}.
  unfold valid_hoare_triple in *.
  intros P Q b c Hstart Hinv st st'' Heval HP.
  remember <{repeat c until b end}> as Hcommand.
  induction Heval; try discriminate.
  - admit.
  - apply (IHHeval2 HeqHcommand).
  -
    Abort.
```

Hoare_repeat : failed proof

```
apply (IHHeval2 HeqHcommand).
```

```
P, Q : Assertion
b : bexp
c : com
Hstart : ∀ st st' : state, st =[ c ]⇒ st' → P st → Q st'
Hinv : ∀ st st' : state, st =[ c ]⇒ st' → Q st ∧ ~ b st → Q st'
b0 : bexp
c0 : com
HeqHcommand : <{ repeat c0 until b0 end }> = <{ repeat c until b end }>
st, st', st'' : state
Heval1 : st =[ c0 ]⇒ st'
H : beval st' b0 = false
Heval2 : st' =[ repeat c0 until b0 end ]⇒ st''
HP : P st
IHHeval1 : c0 = <{ repeat c until b end }> → P st → Q st' ∧ b st'
IHHeval2 : <{ repeat c0 until b0 end }> = <{ repeat c until b end }> →
           P st' → Q st'' ∧ b st''
______________________________________
Q st'' ∧ b st''
```

Hoare_repeat : failed proof

```
P, Q : Assertion
b : bexp
c : com
Hstart : ∀ st st' : state, st =[ c ]⇒ st' → P st → Q st'
Hinv : ∀ st st' : state, st =[ c ]⇒ st' → Q st ∧ ~ b st → Q st'
b0 : bexp
c0 : com
HeqHcommand : <{ repeat c0 until b0 end }> = <{ repeat c until b end }>
st, st', st'' : state
Heval1 : st =[ c0 ]⇒ st'
H : beval st' b0 = false
Heval2 : st' =[ repeat c0 until b0 end ]⇒ st''
HP : P st
IHHeval1 : c0 = <{ repeat c until b end }> → P st → Q st' ∧ b st'
IHHeval2 : <{ repeat c0 until b0 end }> = <{ repeat c until b end }> →
           P st' → Q st'' ∧ b st''
______________________________________
P st'
```

Hoare_repeat : failed proof

What we have:

```
{{ P }} c {{ Q }}   (Assumption)
{{ Q /\ ~b }} c {{ Q }}  (Assumption)
{{ P }} repeat c until b end {{ Q }}  (IH)

Goal: {{ P }} c ; repeat c until b end {{ Q }}
 ‘=>’ {{ Q }} repeat c until b end {{ Q }}
```

递归假设不够通用，以至于无法证明目标。

Hoare_repeat : key idea

```
Lemma hoare_repeat :
  ∀ P Q (b : bexp) c,
    {{ P }} c {{ Q }} →
    {{ Q ∧ ~b }} c {{ Q }} →
    {{ P }} repeat c until b end {{ Q ∧ b }}.
  unfold valid_hoare_triple in *.
  intros P Q b c Hstart Hinv st st'' Heval HP.
  inversion Heval; try subst; try discriminate.
```

先将目标进行一步 inversion ，得到

{{ Q }} repeat c until b end {{ Q /\ b}}

再进行 induction

Hoare_repeat : key idea

```
Lemma hoare_repeat :
  ∀ P Q (b : bexp) c,
    {{ P }} c {{ Q }} →
    {{ Q ∧ ~b }} c {{ Q }} →
    {{ P }} repeat c until b end {{ Q ∧ b }}.
  unfold valid_hoare_triple in *.
  intros P Q b c Hstart Hinv st st'' Heval HP.
  inversion Heval; try subst; try discriminate.

  - split; try assumption.
    eapply Hstart; eassumption.
  - assert (HQ : Q st') by (eapply Hstart; eassumption).
    clear Heval HP st H1 Hstart.
    remember <{repeat c until b end}> as Hcommand.
    induction H5; try discriminate.
    + inversion HeqHcommand; subst.
      split; try assumption.
      apply (Hinv _ _ H5).
      split; try assumption.
      congruence.
    + inversion HeqHcommand; subst.
      apply IHceval2; try assumption.
      apply (Hinv _ _ H5_).
      split; try assumption.
      congruence.
Qed.
```

Proof by reflection : Motivation

- 对于一些易于处理的目标，我们希望能利用 Coq 的计算能力来简化目标。

```
Example example₁:
  ⊤ ∧ (⊥ ∨ ⊤).
  let temp := reify (⊤ ∧ (⊥ ∨ ⊤)) in pose (goal_ast := temp).
  apply (bool_prop_reflect goal_ast).
  reflexivity.
Qed.
```

- 对这种目标，正常的做法是利用一系列 tactics 来化简从而证明。
- 我们希望 Coq 能自动算出来它是对的.

Proof by reflection : key idea

- 通过把不利于操作的 Prop 变成利于操作的语法树。

```
Inductive bool_ast : Type :=
| BTrue
| BFalse
| BAnd (b₁ b₂ : bool_ast)
| BOr (b₁ b₂ : bool_ast)
.
```

Proof by reflection : key idea

- 通过语法树上的计算来化简 Prop

```
Fixpoint bool_ast_denote (b : bool_ast) : 𝔹 :=
  match b with
   | BTrue ⇒ true
   | BFalse ⇒ false
   | BAnd b1 b2 ⇒ (bool_ast_denote b1) && (bool_ast_denote b2)
   | BOr b1 b2 ⇒ (bool_ast_denote b1) || (bool_ast_denote b2)
  end.
```

Proof by reflection : key idea

- 同时需要保证语法树上的计算确实是正确的计算。

```
Fixpoint prop_ast_denote (b : bool_ast) : P :=
  match b with
  | BTrue ⇒ ⊤
  | BFalse ⇒ ⊥
  | BAnd b₁ b₂ ⇒ (prop_ast_denote b₁) ∧ (prop_ast_denote b₂)
  | BOr b₁ b₂ ⇒ (prop_ast_denote b₁) ∨ (prop_ast_denote b₂)
  end.

Theorem bool_prop_reflect :
  ∀ b : bool_ast,
    bool_ast_denote b = true →
    prop_ast_denote b.
```

Proof by reflection : bonus

- 如何利用 proof by reflection 来证明:

```
Example add_assoc :
  ∀ a b c d e,
    a + (b + c) + (d + e) = a + b + c + d + e.
```

- (Hard) 可以发现上述目标仅使用加法结合律即可完成目标，那对于同时包含加法结合律和加法交换律的目标呢?
- Acknowledgement & Further reading :
  - Chapter 15, Certified Programming with Dependent Types by Adam Chlipala

Q&A

- Q1: 什么是 Calculus of Inductive Constructions?
- A1: 同时包含:
  - (Simply-typed lambda calculus) 值能作用在值上 : add 0 1
  - (Polymorphism) 值能作用在类型上 : cons Nat 0 nil
  - (Type Operator) 类型能作用在类型上 : List Nat
  - (Dependent type)类型能作用在值上 : Vec 0 nat (固定长度的列表)

  (以上称为 Calculus of Constructions)
  - 以及 Inductive datatype

的类型系统。

Q&A

- Bonus Q1: 为什么 Inductive datatype 要单独提及?
- A1: 在 Coq 中尝试以下 Inductive 定义

```
Inductive wrong : Type :=
| TApp (f : wrong → wrong)
.
```

Q&A

Further reading：

- Types and Programming Languages by Benjamin Pierce
- self-contained reader-friendly introduction to CoC by Helmut Brandl
    - https://hbr.github.io/Lambda-Calculus/cc-tex/cc.pdf

Q&A

- Q2 : Rust unsafe 代码验证以及内核代码验证?
- A2 :
  - Rust unsafe 代码验证 (Jung, Ralf, et al. "RustBelt: Securing the foundations of the Rust programming language." Proceedings of the ACM on Programming Languages 2.POPL (2017): 1-34)
  - 内核代码验证：欢迎课后与我联系，欢迎加入程序设计语言研究室。