指称语义

1 简单表达式的指称语义

指称语义是一种定义程序行为的方式。在极简的 SimpleWhile 语言中，整数类型表达式中只有整数常量、变量、加法、减法与乘法运算。

```
EI :: = N | V | EI + EI | EI - EI | EI * EI
```

我们约定其中整数变量的值、整数运算的结果都是没有范围限制的。基于这一约定，我们可以如下定义表达式`e`在程序状态`st`上的值。

首先定义程序状态集合：

$$\text{state} \triangleq \text{var\_name} \to \mathbb{Z}$$

```
Definition state: Type := var_name -> Z.
```

- $[\![n]\!](s) = n$
- $[\![x]\!](s) = s(x)$
- $[\![e_1 + e_2]\!](s) = [\![e_1]\!](s) + [\![e_2]\!](s)$
- $[\![e_1 - e_2]\!](s) = [\![e_1]\!](s) - [\![e_2]\!](s)$
- $[\![e_1 * e_2]\!](s) = [\![e_1]\!](s) * [\![e_2]\!](s)$

其中 $s \in \text{state}$。

下面使用 Coq 递归函数定义整数类型表达式的行为。

```
Fixpoint eval_expr_int (e: expr_int) (s: state) : Z :=
  match e with
  | EConst n => n
  | EVar X => s X
  | EAdd e1 e2 => eval_expr_int e1 s + eval_expr_int e2 s
  | ESub e1 e2 => eval_expr_int e1 s - eval_expr_int e2 s
  | EMul e1 e2 => eval_expr_int e1 s * eval_expr_int e2 s
  end.
```

下面是两个具体的例子。

```
Example eval_example1: forall (s: state),
  s "x" = 1 ->
  s "y" = 2 ->
  eval_expr_int [["x" + "y"]] s = 3.
Proof. intros. simpl. rewrite H, H0. reflexivity. Qed.
```

```
Example eval_example2: forall (s: state),
  s "x" = 1 ->
  s "y" = 2 ->
  eval_expr_int [["x" * "y" + 1]] s = 3.
Proof. intros. simpl. rewrite H, H0. reflexivity. Qed.
```

2 行为等价

基于整数类型表达式的语义定义`eval_expr_int`，我们可以定义整数类型表达式之间的行为等价（亦称语义等价）：两个表达式`e1`与`e2`是等价的当且仅当它们在任何程序状态上的求值结果都相同。

```
Definition expr_int_equiv (e1 e2: expr_int): Prop :=
  forall st, eval_expr_int e1 st = eval_expr_int e2 st.

Notation "e1 '~=~' e2" := (expr_int_equiv e1 e2)
  (at level 69, no associativity).
```

下面是一些表达式语义等价的例子。

```
Example expr_int_equiv_sample:
  [["x" + "x"]] ~=~ [["x" * 2]].
Proof.
  intros.
  unfold expr_int_equiv.
```

上面的`unfold`指令表示展开一项定义，一般用于非递归的定义。

```
  intros.
  simpl.
  lia.
Qed.
```

```
Lemma zero_plus_equiv: forall (a: expr_int),
  [[0 + a]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma plus_zero_equiv: forall (a: expr_int),
  [[a + 0]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma minus_zero_equiv: forall (a: expr_int),
  [[a - 0]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma zero_mult_equiv: forall (a: expr_int),
  [[0 * a]] ~=~ 0.
(* 证明详见Coq源代码。 *)

Lemma mult_zero_equiv: forall (a: expr_int),
  [[a * 0]] ~=~ 0.
(* 证明详见Coq源代码。 *)

Lemma const_plus_const: forall n m: Z,
  [[EConst n + EConst m]] ~=~ EConst (n + m).
(* 证明详见Coq源代码。 *)

Lemma const_minus_const: forall n m: Z,
  [[EConst n - EConst m]] ~=~ EConst (n - m).
(* 证明详见Coq源代码。 *)

Lemma const_mult_const: forall n m: Z,
  [[EConst n * EConst m]] ~=~ EConst (n * m).
(* 证明详见Coq源代码。 *)
```

下面定义一种简单的语法变换—常量折叠—并证明其保持语义等价性。所谓常量折叠指的是将只包含常量而不包含变量的表达式替换成为这个表达式的值。

```
Fixpoint fold_constants (e : expr_int) : expr_int :=
  match e with
  | EConst n    => EConst n
  | EVar x      => EVar x
  | EAdd e1 e2  =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 + n2)
      | _, _ => EAdd (fold_constants e1) (fold_constants e2)
      end
  | ESub e1 e2 =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 - n2)
      | _, _ => ESub (fold_constants e1) (fold_constants e2)
    end
  | EMul e1 e2 =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 * n2)
      | _, _ => EMul (fold_constants e1) (fold_constants e2)
    end
  end.
```

这里我们可以看到，Coq 中`match`的使用是非常灵活的。(1) 我们不仅可以对一个变量的值做分类讨论，还可以对一个复杂的 Coq 式子的取值做分类讨论；(2) 我们可以对多个值同时做分类讨论；(3) 我们可以用下划线表示`match`的缺省情况。下面是两个例子：

```
Example fold_constants_ex1:
    fold_constants [[(1 + 2) * "k"]] = [[3 * "k"]].
Proof. intros. reflexivity. Qed.
```

注意，根据我们的定义，`fold_constants`并不会将`0 + "y"`中的`0`消去。

```
Example fold_expr_int_ex2 :
  fold_constants [["x" - ((0 * 6) + "y")]] = [["x" - (0 + "y")]].
Proof. intros. reflexivity. Qed.
```

下面我们在 Coq 中证明，`fold_constants`保持表达式行为不变。

```
Theorem fold_constants_sound : forall a,
  fold_constants a ~=~ a.
Proof.
  unfold expr_int_equiv. intros.
  induction a.
```

常量的情况

```
  + simpl.
    reflexivity.
```

变量的情况

```
  + simpl.
    reflexivity.
```

加号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
```

减号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
```

乘号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
Qed.
```

3 利用高阶函数定义指称语义

```
Definition add_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s + D2 s.
```

```
Definition sub_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s - D2 s.
```

```
Definition mul_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s * D2 s.
```

下面是用于类型查询的`Check`指令。

```
Check add_sem.
```

可以看到`add_sem`的类型是`(state -> Z) -> (state -> Z) -> state -> Z`，这既可以被看做一个三元函数，也可以被看做一个二元函数，即函数之间的二元函数。

基于上面高阶函数，可以重新定义表达式的指称语义。

```
Definition const_sem (n: Z) (s: state): Z := n.
Definition var_sem (X: var_name) (s: state): Z := s X.
```

```
Fixpoint eval_expr_int (e: expr_int) : state -> Z :=
  match e with
  | EConst n =>
      const_sem n
  | EVar X =>
      var_sem X
  | EAdd e1 e2 =>
      add_sem (eval_expr_int e1) (eval_expr_int e2)
  | ESub e1 e2 =>
      sub_sem (eval_expr_int e1) (eval_expr_int e2)
  | EMul e1 e2 =>
      mul_sem (eval_expr_int e1) (eval_expr_int e2)
  end.
```

4 更多高阶函数的例子

在 Coq 中，一个函数可以以函数作为参数，也可以以以函数作为返回值，这样的函数称为高阶函数。下面是一个典型的例子。

```
Definition doit3times {X: Type} (f: X -> X) (n: X): X :=
  f (f (f n)).
```

这里，`f`这个参数本身也是一个函数（从`X`到`X`的函数）而`doit3times`则把`f`在`n`上作用了 3 次。

```
Definition minustwo (x: Z): Z := x - 2.
```

习题 1.

- `doit3times minustwo 9`的值是多少？**参考答案.** 3
- `doit3times minustwo (doit3times minustwo 9)`的值是多少？**参考答案.** -3
- `doit3times (doit3times minustwo) 9`的值是多少？**参考答案.** -9
- `doit3times doit3times minustwo 9`的值是多少？**参考答案.** -45

Coq 中允许用户使用`fun`关键字表示匿名函数，例如：

```
Example fact_doit3times_anon1:
  doit3times (fun x => x - 2) 9 = 3.
Proof. reflexivity. Qed.
```

```
Example fact_doit3times_anon2:
  doit3times (fun x => x * x) 2 = 256.
Proof. reflexivity. Qed.
```

这里`fun x => x - 2`与之前定义的`minustwo`是相同的，而`fun x => x * x`则表示了平方这样一个函数。

习题 2.

- `doit3times (fun x => - x) 5`的值是多少？**参考答案.** -5
- `doit3times (fun f x y => f y x) (fun x y => x - y) 1 2`的值是多少？**参考答案.** 1
- 如果`Func_add`定义为`fun f g x => f x + g x`那么`doit3times (Func_add minustwo) (fun x => x * x)`相当于什么函数？**参考答案.** `x * x + x * 3 - 6`
- `doit3times ((fun x y => y * y - x * y + x * x) 1) 1`的值是多少？**参考答案.** 1

5 布尔表达式语义

对于任意布尔表达式 e，我们规定它的语义 $[\![e]\!]$ 是一个程序状态到真值的函数，表示表达式 e 在各个程序状态上的求值结果。

- $[\![\text{TRUE}]\!](s) = \mathbf{T}$
- $[\![\text{FALSE}]\!](s) = \mathbf{F}$
- $[\![e_1 < e_2]\!](s)$ 为真当且仅当 $[\![e_1]\!](s) < [\![e_2]\!](s)$
- $[\![e_1 \&\& e_2]\!](s) = [\![e_1]\!](s)$ and $[\![e_2]\!](s)$
- $[\![!e_1]\!](s) = \text{not } [\![e_1]\!](s)$

在 Coq 中可以如下定义：

```
Definition true_sem (s: state): bool := true.
```

```
Definition false_sem (s: state): bool := false.
```

```
Definition lt_sem (D1 D2: state -> Z) s: bool :=
  Z.ltb (D1 s) (D2 s).
```

```
Definition and_sem (D1 D2: state -> bool) s: bool :=
  andb (D1 s) (D2 s).
```

```
Definition not_sem (D: state -> bool) s: bool :=
  negb (D s).
```

```
Fixpoint eval_expr_bool (e: expr_bool): state -> bool :=
  match e with
  | ETrue =>
      true_sem
  | EFalse =>
      false_sem
  | ELt e1 e2 =>
      lt_sem (eval_expr_int e1) (eval_expr_int e2)
  | EAnd e1 e2 =>
      and_sem (eval_expr_bool e1) (eval_expr_bool e2)
  | ENot e1 =>
      not_sem (eval_expr_bool e1)
  end.
```

6 程序语句的语义

$(s_1, s_2) \in [\![c]\!]$ 当且仅当从 s_1 状态开始执行程序 c 会以程序状态 s_2 终止。

6.1 赋值语句的语义

$$[\![x = e]\!] = \{(s_1, s_2) \mid s_2(x) = [\![e]\!](s_1), \text{for any } y \in \text{var\_name}, \text{ if } x \neq y,\ s_1(y) = s_2(y)\}$$

6.2 空语句的语义

$$[\![\text{SKIP}]\!] = \{(s_1, s_2) \mid s_1 = s_2\}$$

6.3 顺序执行语句的语义

$$[\![c_1; c_2]\!] = [\![c_1]\!] \circ [\![c_2]\!] = \{(s_1, s_3) \mid (s_1, s_2) \in [\![c_1]\!], (s_2, s_3) \in [\![c_2]\!]\}$$

6.4 条件分支语句的语义

$$\begin{aligned}[\![\text{if } (e) \text{ then}\{c_1\} \text{ else } \{c_2\}]\!] = \quad & \left(\{(s_1, s_2) \mid [\![e]\!](s_1) = \mathbf{T}\} \cap [\![c_1]\!]\right) \cup \\ & \left(\{(s_1, s_2) \mid [\![e]\!](s_1) = \mathbf{F}\} \cap [\![c_2]\!]\right)\end{aligned}$$

这又可以改写为：

$$[\![\text{if } (e) \text{ then}\{c_1\} \text{ else } \{c_2\}]\!] = \text{test\_true}([\![e]\!]) \circ [\![c_1]\!] \cup \text{test\_false}([\![e]\!]) \circ [\![c_2]\!]$$

其中，

$$\text{test\_true}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{T}, s_1 = s_2\}$$

$$\text{test\_false}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{F}, s_1 = s_2\}$$

6.5 循环语句的语义

定义方式一：

$$\text{iter}_0(X, R) = \text{test\_false}(X)$$

$$\text{iter}_{n+1}(X, R) = \text{test\_true}(X) \circ R \circ \text{iter}_n(X, R)$$

$$[\![\text{while } (e) \text{ do } \{c\}]\!] = \bigcup_{n \in \mathbb{N}} \text{iter}_n([\![e]\!], [\![c]\!])$$

定义方式二：

$$\text{iter}_{<0}(X, R) = \varnothing$$

$$\text{iter}_{<n+1}(X, R) = \text{test\_true}(X) \circ R \circ \text{iter}_{<n}(X, R) \cup \text{test\_false}(X)$$

$$[\![\text{while } (e) \text{ do } \{c\}]\!] = \bigcup_{n \in \mathbb{N}} \text{iter}_{<n}([\![e]\!], [\![c]\!])$$

7 Coq 中的集合与关系

在 Coq 中往往使用`X: A -> Prop`来表示某类型`A`中元素构成的集合`X`。字面上看，这里的`A -> Prop`表示`X`是一个从`A`中元素到命题的映射，这也相当于说`X`是一个关于`A`中元素性质。对于每个`A`中元素`a`而言，`a`符合该性质`X`等价于`a`对应的命题`X a`为真，又等价于`a`是集合`X`的元素。

类似的，`R: A -> B -> Prop`也用来表示`A`与`B`中元素之间的二元关系。

本课程提供的 SetsClass 库中提供了有关集合的一系列定义。例如：

- 空集：用 `∅` 或者一堆方括号表示，定义为`Sets.empty`；
- 单元集：用一对方括号表示，定义为`Sets.singleton`；
- 并集：用`∪`表示，定义为`Sets.union`；
- 交集：用`∩`表示，定义为`Sets.intersect`；
- 一列集合的并：用`⋃`表示，定义为`Sets.omega_union`；
- 一列集合的交：用`⋂`表示，定义为`Sets.omega_intersect`；
- 集合相等：用`==`表示，定义为`Sets.equiv`；
- 元素与集合关系：用`∈`表示，定义为`Sets.In`；
- 子集关系：用`⊆`表示，定义为`Sets.included`；
- 二元关系的连接：用`∘`表示，定义为`Rels.concat`；
- 等同关系：定义为`Rels.id`；
- 测试关系：定义为`Rels.test`。

在 CoqIDE 中，你可以利用 CoqIDE 对于 unicode 的支持打出特殊字符：

- 首先，在打出特殊字符的 latex 表示法；
- 再按 shift+ 空格键；
- latex 表示法就自动转化为了相应的特殊字符。

例如，如果你需要打出符号`∈`，请先在输入框中输入`\in`，当光标紧跟在`n`这个字符之后的时候，按 shift+ 空格键即可。例如，下面是两个关于集合的命题：

```
Check forall A (X: A -> Prop), X ∪ ∅ == X.

Check forall A B (X Y: A -> B -> Prop), X ∪ Y ⊆ X.
```

由于集合以及集合间的运算是基于 Coq 中的命题进行定义的，集合相关性质的证明也可以规约为与命题有关的逻辑证明。例如，我们想要证明，交集运算具有交换律：

```
Lemma Sets_intersect_comm: forall A (X Y: A -> Prop),
  X ∩ Y == Y ∩ X.
Proof.
  intros.
```

下面一条命令`sets_unfold`是 SetsClass 库提供的自动证明指令，它可以将有关集合的性质转化为有关命题的性质。

```
  sets_unfold.
```

原本要证明的关于交集的性质现在就转化为了：`forall a : A, X a /\ Y a <-> Y a /\ X a` 其中`forall`就是逻辑中『任意』的意思；`/\`之前我们已经了解，它表示『并且』的意思；`<->`表示『当且仅当』的意思；`X a`可以念做性质`X`对于`a`成立，也可以理解为`a`是集合`X`的元素。

我们稍后再来完成相关性质的证明。在 Coq 中，要放弃当前的证明，可以用下面的`Abort`指令。

```
Abort.
```

下面是一条关于并集运算的性质。

```
Lemma Sets_included_union1: forall A (X Y: A -> Prop),
  X ⊆ X ∪ Y.
Proof.
  intros.
  sets_unfold.
```

经过转化，要证明的结论是：`forall a : A, X a -> X a \/ Y a`。这里，`\/`表示『或者』；`->`表示推出，也可以念做『如果... 那么...』。

```
Abort.
```

下面是一条关于二元运算的性质。

```
Lemma Rels_concat_assoc: forall A (X Y Z: A -> A -> Prop),
  (X ∘ Y) ∘ Z == X ∘ (Y ∘ Z).
Proof.
  intros.
  unfold_RELS_tac.
```

关于二元关系的性质，要使用`unfold_RELS_tac`展开成为基于逻辑的定义。

```
Abort.
```

SetsClass 库中提供了一系列有关集合运算的性质的证明。未来大家在证明中既可以使用`sets_unfold`与`unfold_RELS_tac`关于集合运算的命题转化为关于逻辑的命题，也可以直接使用下面这些性质完成证明。

```
  Sets_equiv_Sets_included:
    forall x y, x == y <-> x ⊆ y /\ y ⊆ x;
  Sets_empty_included:
    forall x, ∅ ⊆ x;
  Sets_included_full:
    forall x, x ⊆ Sets.full;
  Sets_intersect_included1:
    forall x y, x ∩ y ⊆ x;
  Sets_intersect_included2:
    forall x y, x ∩ y ⊆ y;
  Sets_included_intersect:
    forall x y z, x ⊆ y -> x ⊆ z -> x ⊆ y ∩ z;
  Sets_included_union1:
    forall x y, x ⊆ x ∪ y;
  Sets_included_union2:
    forall x y, y ⊆ x ∪ y;
  Sets_union_included_strong2:
    forall x y z u,
       x ∩ u ⊆ z -> y ∩ u ⊆ z -> (x ∪ y) ∩ u ⊆ z;
```

```
  Sets_included_omega_union:
    forall xs n, xs n ⊆ ⋃ xs;
  Sets_omega_union_included:
    forall xs y, (forall n, xs n ⊆ y) -> ⋃ xs ⊆ y;
  Sets_omega_intersect_included:
    forall xs n, ⋂ xs ⊆ xs n;
  Sets_included_omega_intersect:
    forall xs y, (forall n : nat, y ⊆ xs n) -> y ⊆ ⋂ xs;
```

```
  Rels_concat_union_distr_r:
    forall x1 x2 y,
      (x1 ∪ x2) ∘ y == (x1 ∘ y) ∪ (x2 ∘ y);
  Rels_concat_union_distr_l:
    forall x y1 y2,
      x ∘ (y1 ∪ y2) == (x ∘ y1) ∪ (x ∘ y2);
  Rels_concat_mono:
    forall x1 x2,
      x1 ⊆ x2 ->
      forall y1 y2,
        y1 ⊆ y2 ->
        x1 ∘ y1 ⊆ x2 ∘ y2;
  Rels_concat_assoc:
    forall x y z,
      (x ∘ y) ∘ z == x ∘ (y ∘ z);
```

8 Coq 中关于逻辑的证明

在 Coq 中可以用与、或、非、如果-那么、存在以及任意来描述复杂的命题，并且证明相关性质。

8.1 逻辑命题『真』的证明

我们不需要任何前提就可以推出`True`。在 Coq 标准库中，`I`是`True`的一个证明，我们可以用`exact I`来证明`True`。

```
Example proving_True_1: 1 < 2 -> True.
Proof.
  intros.
  exact I.
Qed.
```

```
Example proving_True_2: 1 > 2 -> True.
Proof.
  intros.
  exact I.
Qed.
```

8.2 关于『并且』的证明

要证明『某命题并且某命题』成立，可以在 Coq 中使用`split`证明指令进行证明。该指令会将当前的证明目标拆成两个子目标。

```
Lemma True2: True /\ True.
Proof.
  split.
  + exact I.
  + exact I.
Qed.
```

下面证明一个关于`/\`的一般性结论：

```
Lemma and_intro : forall A B : Prop, A -> B -> A /\ B.
Proof.
  intros A B HA HB.
  split.
```

下面的`apply`指令表示在证明中使用一条前提，或者使用一条已经经过证明的定理或引理。

```
  + apply HA.
  + apply HB.
Qed.
```

习题 3.

```
Example and_exercise :
  forall n m : Z, n + 2*m = 10 -> 2*n + m = 5 -> n = 0 /\ True.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

参考答案.

```
Proof.
  intros.
  split.
  + lia.
  + exact I.
Qed.
```

如果当前一条前提假设具有『某命题并且某命题』的形式，我们可以在 Coq 中使用`destruct`指令将其拆分成为两个前提。

```
Lemma proj1 : forall P Q : Prop,
  P /\ Q -> P.
Proof.
  intros.
  destruct H as [HP HQ].
  apply HP.
Qed.
```

`destruct`指令也可以不指名拆分后的前提的名字，Coq 会自动命名。

```
Lemma proj2 : forall P Q : Prop,
  P /\ Q -> Q.
Proof.
  intros.
  destruct H.
  apply H0.
Qed.
```

当前提与结论中，都有`/\`的时候，我们就既需要使用`split`指令，又需要使用`destruct`指令。

```
Theorem and_commut : forall P Q : Prop,
  P /\ Q -> Q /\ P.
Proof.
  intros.
  destruct H as [HP HQ].
  split.
  - apply HQ.
  - apply HP.
Qed.
```

习题 4.

```
Theorem and_assoc : forall P Q R : Prop,
  P /\ (Q /\ R) -> (P /\ Q) /\ R.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

参考答案.

```
Proof.
  intros.
  destruct H.
  destruct H0.
  split.
  + split.
    - apply H.
    - apply H0.
  + apply H1.
Qed.
```

8.3 关于『或』的证明

『或』是另一个重要的逻辑连接词。如果『或』出现在前提中，我们可以用 Coq 中的 `destruct` 指令进行分类讨论。

```
Lemma or_example :
  forall n m : Z, n = 0 \/ m = 0 -> n * m = 0.
Proof.
  intros.
  destruct H as [H | H].
  + rewrite H.
    lia.
  + rewrite H.
    lia.
Qed.
```

在上面的例子中，我们对于形如 `A \/ B` 的前提进行分类讨论。要证明 `A \/ B` 能推出原结论，就需要证明 `A` 与 `B` 中的任意一个都可以推出原结论。下面是一个一般性的结论.

```
Lemma or_example2 :
  forall P Q R: Prop, (P -> R) -> (Q -> R) -> (P \/ Q -> R).
Proof.
  intros.
  destruct H1 as [HP | HQ].
  + apply H.
```

注意，`apply` 指令不一定要前提与结论完全吻合才能使用。此处，只要 `H` 中推导的结果与待证明的结论一致，就可以使用 `apply H`。

```
    apply HP.
  + apply H0 in HQ.
```

`apply` 指令还可以在前提中做推导，不过这时需要使用 `apply ... in` 这一语法。

```
    apply HQ.
Qed.
```

相反的，如果要证明一条形如 `A \/ B` 的结论整理，我们就只需要证明 `A` 与 `B` 两者之一成立就可以了。在 Coq 中的指令是：`left` 与 `right`。例如，下面是选择左侧命题的例子。

```
Lemma or_introl : forall A B : Prop, A -> A \/ B.
Proof.
  intros.
  left.
  apply H.
Qed.
```

下面是选择右侧命题的例子。

```
Lemma or_intror : forall A B : Prop, B -> A \/ B.
Proof.
  intros.
  right.
  apply H.
Qed.
```

下面性质请各位自行证明。

习题 5.

```
Theorem or_commut : forall P Q : Prop,
  P \/ Q  -> Q \/ P.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

参考答案.

```
Proof.
  intros.
  destruct H as [HP | HQ].
  + right.
    apply HP.
  + left.
    apply HQ.
Qed.
```

8.4 关于『如果... 那么...』的证明

事实上，在之前的例子中，我们已经多次证明有关`->`的结论了。下面我们在看几个例子，并额外介绍几条 Coq 证明指令。

下面的证明中，`pose proof`表示推导出一个新的结论，并将其用作之后证明中的前提。

```
Theorem modus_ponens: forall P Q: Prop,
  P /\ (P -> Q) -> Q.
Proof.
  intros.
  destruct H.
```

将`H0: P -> Q`作用在`H: P`上，我们就可以得出一个新结论：`Q`。

```
  pose proof H0 H.
  apply H1.
Qed.
```

下面我们换一种方法证明。`revert`证明指令可以看做`intros`的反操作。

```
Theorem modus_ponens_alter1: forall P Q: Prop,
  P /\ (P -> Q) -> Q.
Proof.
  intros.
  destruct H.
```

下面`revert`指令将前提中的`P`又放回了『结论中的前提』中去。

```
  revert H.
  apply H0.
Qed.
```

下面我们再换一种方式证明，`specialize`指令与`apply ... in`指令的效果稍有不同。

```
Theorem modus_ponens_alter2: forall P Q: Prop,
  P /\ (P -> Q) -> Q.
Proof.
  intros.
  destruct H.
  specialize (H0 H).
  apply H0.
Qed.
```

另外，我们可以直接使用`exact`指令，这个指令的效果像是`pose proof`或者`specialize`与`apply`的组合。

```
Theorem modus_ponens_alter3: forall P Q: Prop,
  P /\ (P -> Q) -> Q.
Proof.
  intros.
  destruct H.
  exact (H0 H).
Qed.
```

8.5 关于『否定』与『假』的证明

在 Coq 中 [ ] 表示否定，`False`表示假。如果前提为假，那么，矛盾推出一切。在 Coq 中，这可以用`contradiction`指令或`destruct`指令完成证明。

```
Theorem ex_falso_quodlibet : forall (P: Prop),
  False -> P.
Proof.
  intros.
  contradiction.
Qed.
```

```
Theorem ex_falso_quodlibet_alter : forall (P: Prop),
  False -> P.
Proof.
  intros.
  destruct H.
Qed.
```

`contradiction`也可以用于`P`与`~ P`同时出现在前提中的情况：

```
Theorem contradiction_implies_anything : forall P Q : Prop,
  (P /\ ~ P) -> Q.
Proof.
  intros.
  destruct H.
  contradiction.
Qed.
```

除了`P`与`~ P`不能同时为真之外，他们也不能同时为假，或者说，他们中至少有一个要为真。这是 Coq 标准库中的`classic`。

```
Check classic.
```

它说的是：`forall P : Prop, P \/ ~ P`。下面我们利用它做一些证明。

```
Theorem double_neg_elim : forall P : Prop,
  ~ ~ P -> P.
Proof.
  intros.
  pose proof classic P.
  destruct H0.
  + apply H0.
  + contradiction.
Qed.
```

习题 6.

```
Theorem not_False :
  ~ False.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

参考答案.

```
Proof.
  pose proof classic False.
  destruct H.
  + contradiction.
  + apply H.
Qed.
```

习题 7.

```
Theorem double_neg_intro : forall P : Prop,
  P -> ~ ~ P.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

参考答案.

```
Proof.
  intros.
  pose proof classic (~ P).
  destruct H0.
  + contradiction.
  + apply H0.
Qed.
```

8.6 关于『当且仅当』的证明

在 Coq 中，`<->` 符号对应的定义是 `iff`，其将 `P <-> Q` 定义为 `(P -> Q) /\ (Q -> P)` 因此，要证明关于『当且仅当』的性质，首先可以使用其定义进行证明。

```
Theorem iff_refl: forall P: Prop, P <-> P.
Proof.
  intros.
  unfold iff.
  split.
  + intros.
    apply H.
  + intros.
    apply H.
Qed.
```

```
Theorem iff_imply: forall P Q: Prop, (P <-> Q) -> (P -> Q).
Proof.
  intros P Q H.
  unfold iff in H.
  destruct H.
  exact H.
Qed.
```

当某前提假设具有形式 `P <-> Q`，那我们也可以使用 `apply` 指令进行证明。

```
Theorem iff_imply_alter: forall P Q: Prop, (P <-> Q) -> (P -> Q).
Proof.
  intros.
  apply H.
  apply H0.
Qed.
```

另外，`rewrite` 指令也可以使用形如 `P <-> Q` 的等价性前提。

```
Theorem iff_imply_alter2: forall P Q: Prop, (P <-> Q) -> (P -> Q).
Proof.
  intros.
  rewrite <- H.
  apply H0.
Qed.
```

8.7 关于『存在』的证明

当待证明结论形为：存在一个 `x` 使得...，那么可以用 `exists` 指明究竟哪个 `x` 使得该性质成立。

```
Lemma four_is_even : exists n, 4 = n + n.
Proof.
  exists 2.
  lia.
Qed.
```

```
Lemma six_is_not_prime: exists n, 2 <= n < 6 /\ exists q, n * q = 6.
Proof.
  exists 2.
  split.
  + lia.
  + exists 3.
    lia.
Qed.
```

当某前提形为：存在一个`x`使得…，那么可以使用 Coq 中的`destruct`指令进行证明。这一证明指令相当于数学证明中的：任意给定一个这样的`x`。

```
Theorem exists_example : forall n,
  (exists m, n = 4 + m) ->
  (exists o, n = 2 + o).
Proof.
  intros.
  destruct H as [m H].
  exists (2 + m).
  lia.
Qed.
```

习题 8.

```
Theorem dist_exists_and : forall (X: Type) (P Q: X -> Prop),
  (exists x, P x /\ Q x) -> (exists x, P x) /\ (exists x, Q x).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

参考答案.

```
Proof.
  intros.
  destruct H as [x [HP HQ]].
  split.
  + exists x.
    apply HP.
  + exists x.
    apply HQ.
Qed.
```

习题 9.

```
Theorem exists_exists : forall (X Y: Type) (P: X -> Y -> Prop),
  (exists x y, P x y) <-> (exists y x, P x y).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

参考答案.

```
Proof.
  intros.
  split; intros.
  + destruct H as [x [y ?]].
    exists y, x.
    exact H.
  + destruct H as [y [x ?]].
    exists x, y.
    exact H.
Qed.
```

8.8 关于『任意』的证明

关于『任意』的证明与关于『如果... 那么...』的证明是类似的，我们可以灵活使用 `pose proof`，`specialize`，`apply`，`revert` 等指令进行证明。下面是一个简单的例子。

```
Theorem forall_forall : forall (X Y: Type) (P: X -> Y -> Prop),
  (forall x y, P x y) -> (forall y x, P x y).
Proof.
  intros X Y P H.
  intros y x.
  specialize (H x y).
  apply H.
Qed.
```

8.9 关于命题逻辑的自动证明

除了上述证明指令之外，Coq 还提供了 `tauto` 这一自动证明指令。如果当且结论可以完全通过命题逻辑完成证明，那么 `tauto` 就可以自动构造这样的证明。例如：

```
Lemma or_reduce: forall (P Q R: Prop),
  (P /\ R) \/ (Q /\ ~ R) -> P \/ Q.
Proof.
  intros.
  tauto.
Qed.
```

这里，所谓完全通过命题逻辑完成证明，指的是通过对于『并且』、『或』、『非』、『如果... 那么...』、『当且仅当』、『真』与『假』的推理完成证明（注：除此之外，`tauto` 也会将形如 `a = a` 的命题看做 `True`）。下面例子所描述的命题很符合直觉，但是它就不能仅仅使用命题逻辑完成推理，因为他的推理过程中用到了 `forall` 的性质。

```
Lemma forall_and: forall (A: Type) (P Q: A -> Prop),
  (forall a: A, P a /\ Q a) <-> (forall a: A, P a) /\ (forall a: A, Q a).
Proof.
  intros.
```

注意，此处不能使用 `tauto` 直接完成证明。

```
  split.
  + intros.
    split.
    - intros a.
      specialize (H a).
```

此时可以用 `tauto` 完成剩余证明了，另外两个分支也是类似。

```
      tauto.
    - intros a.
      specialize (H a).
      tauto.
  + intros.
    destruct H.
    specialize (H a).
    specialize (H0 a).
    tauto.
Qed.
```

9 在 Coq 中定义程序语句的语义

下面在 Coq 中写出程序语句的指称语义。

```
Definition skip_sem: state -> state -> Prop :=
  Rels.id.
```

```
Definition asgn_sem
             (X: var_name)
             (D: state -> Z)
             (st1 st2: state): Prop :=
  st2 X = D st1 /\
  forall Y, X <> Y -> st2 Y = st1 Y.
```

```
Definition seq_sem
             (D1 D2: state -> state -> Prop):
  state -> state -> Prop :=
  D1 ∘ D2.
```

```
Definition test_true
             (D: state -> bool):
  state -> state -> Prop :=
  Rels.test (fun st => D st = true).
```

```
Definition test_false
             (D: state -> bool):
  state -> state -> Prop :=
  Rels.test (fun st => D st = false).
```

```
Definition if_sem
             (D0: state -> bool)
             (D1 D2: state -> state -> Prop):
  state -> state -> Prop :=
  (test_true D0 ∘ D1) ∪ (test_false D0 ∘ D2).
```

```
Fixpoint iter_n
           (D0: state -> bool)
           (D1: state -> state -> Prop)
           (n: nat):
  state -> state -> Prop :=
  match n with
  | O => test_false D0
  | S n0 => test_true D0 ∘ D1 ∘ iter_n D0 D1 n0
  end.
```

```
Definition while_sem
             (D0: state -> bool)
             (D1: state -> state -> Prop):
  state -> state -> Prop :=
  ⋃ (iter_n D0 D1).
```

```
Fixpoint iter_lt_n
           (D0: state -> bool)
           (D1: state -> state -> Prop)
           (n: nat):
  state -> state -> Prop :=
  match n with
  | O => ∅
  | S n0 =>
      (test_true D0 ∘ D1 ∘ iter_lt_n D0 D1 n0) ∪
      (test_false D0)
  end.
```

```
Definition while_sem
             (D0: state -> bool)
             (D1: state -> state -> Prop):
  state -> state -> Prop :=
  ⋃ (iter_lt_n D0 D1).
```

下面是程序语句指称语义的递归定义。

```
Fixpoint eval_com (c: com): state -> state -> Prop :=
  match c with
  | CSkip =>
      skip_sem
  | CAsgn X e =>
      asgn_sem X (eval_expr_int e)
  | CSeq c1 c2 =>
      seq_sem (eval_com c1) (eval_com c2)
  | CIf e c1 c2 =>
      if_sem (eval_expr_bool e) (eval_com c1) (eval_com c2)
  | CWhile e c1 =>
      while_sem (eval_expr_bool e) (eval_com c1)
  end.
```