课程信息简介

1 课程内容

这门程序验证课会讲授一系列关于程序正确性的理论知识，并介绍一些实践中较为可行的技术方法。

2 课程评分

本课程的期末总评分分为三个部分。

- 课堂参与 10 分
- 平时作业 50 分
- 大作业 40 分

3 语法与语义

下面三组 C 程序语句中，每一组内的两句程序语句是相同的程序语句吗？第一组：

```
y=x+1; // 代码中的空格更少
```

```
y = x + 1; // 代码中的空格更多
```

第二组：

```
y = (x) + 1; // 代码中的包含多余的括号
```

```
y = x + 1; // 代码中无多余的括号
```

第三组：

```
y = 1 + x;
```

```
y = x + 1;
```

对于描述程序行为与程序正确性的理论而言，像第一组例子中的多余空格与第二组例子中的多余括号并不重要，因此，今后一般就以程序的抽象语法树来定义程序。这样看的话，上面三组程序语句中，第一组与第二组都包含了相同的程序语句。而第三组中的两句程序语句则是不同的程序语句。

程序行为则是完全不同于程序语法的概念。有许多种方法可以描述程序行为。例如我们可以描述程序运行前后的整体效果，我们也可以在此基础上进一步描述程序运行过程中的所有中间步骤。

4 例子：布尔表达式的语法与语义

下面以只包含布尔变元以及与或非的布尔表达式为例，分别定义它们的语法与语义。

首先可以如下定义这类布尔表达式的语法，其中 V 表示布尔变元集合 Σ 中的变元名称。

```
E = V | E && E | E || E | ! E
```

每个布尔表达式 e 的语义（$[\![e]\!]$）可以定义为从真值指派到真值的函数，即定义为这个表达式在每个的真值指派（从布尔变元到真值的映射）上的真值。具体的，假设 $J : \Sigma \to \{\text{true}, \text{false}\}$，那么

- $[\![P]\!](J) = J(P)$，若 $P \in \Sigma$；
- $[\![e_1 \ \&\& \ e_2]\!](J) = [\![\&\&]\!]([\![e_1]\!](J), [\![e_2]\!](J))$；
- $[\![e_1 \ || \ e_2]\!](J) = [\![||]\!]([\![e_1]\!](J), [\![e_2]\!](J))$；
- $[\![!e_1]\!](J) = [\![!]\!]([\![e_1]\!](J))$。

其中，$[\![\&\&]\!]$、$[\![||]\!]$ 与 $[\![!]\!]$ 表示下面的真值函数：

- $[\![\&\&]\!](\text{true}, \text{true}) = \text{true}$，$[\![\&\&]\!](\text{true}, \text{false}) = \text{false}$，
  $[\![\&\&]\!](\text{false}, \text{true}) = \text{false}$，$[\![\&\&]\!](\text{false}, \text{false}) = \text{false}$；
- $[\![||]\!](\text{true}, \text{true}) = \text{true}$，$[\![||]\!](\text{true}, \text{false}) = \text{true}$，
  $[\![||]\!](\text{false}, \text{true}) = \text{true}$，$[\![||]\!](\text{false}, \text{false}) = \text{false}$；
- $[\![!]\!](\text{true}) = \text{false}$，$[\![!]\!](\text{false}) = \text{true}$。

在此基础上我们可以看到，如果 P 与 Q 是两个不同的布尔变元，那么 $P||Q$ 与 $Q||P$ 是不同的布尔表达式（语法树不同），但是具有相同的语义。

5 Coq 证明的例子

这是一个关于群论的证明。首先，我们要定义一个群包含哪些运算。

```
Class GroupOperator: Type := {
  carrier_set: Type;
  zero: carrier_set;
  add: carrier_set -> carrier_set -> carrier_set;
  neg: carrier_set -> carrier_set
}.
```

这里我们可以忽略`Class`、`Type`这些 Coq 保留字，上面这个定义大致说的是：要定义群运算就先要定义一个集合（`carrier_set`），之后一个群应当包含一个单位元（`zero`）、一个二元运算（`add`）以及一个逆元预算（`neg`）。

```
Notation "0" := (zero).
Notation "x + y" := (add x y).
Notation "- x" := (neg x).
```

在 Coq 中，我们可以用零、加号以及负号这些 Notation 来帮助我们表述相关性质。例如：下面就是一个合法的关于群论的命题：

```
Check forall (G: GroupOperator) (x y: carrier_set), x + y = y + x.
```

这里 Coq 的 Check 指令可以理解为检查一个表述语法上是否合法。上面检查的结果是：这是一个合法的 Coq 命题。注意，这只是语法检查，不是证明。下面是群论中的经典证明：从左单位元、左逆元两条性质推出右逆元性质。首先定义：一个群应当具有左单位元、左逆元与结合律这三条性质。

```
Class GroupProperties (G: GroupOperator): Prop := {
  assoc: forall (x y z: carrier_set), (x + y) + z = x + (y + z);
  left_unit: forall (x: carrier_set), 0 + x = x;
  left_inv: forall (x: carrier_set), add (neg x) x = zero
}.
```

其次证明：有上面性质可以推出右逆元性质。

```
Theorem right_inv {G: GroupOperator} {GP: GroupProperties G}:
  (forall (x: carrier_set), x + (- x) = 0).
Proof.
```

在 CoqIDE 中，你可以用 Ctrl+ 向下快捷键让 Coq 检验你的定义与证明。通过检验的代码会变成绿色。进入证明模式后，你会在 CoqIDE 的右边窗口看到现在所有剩余的证明目标。例如，你现在可以看到以下证明目标：

```
  G : GroupOperator
  GP : GroupProperties G
  ============================
  forall x : carrier_set, x + - x = 0
```

这里横线上方的是目前可以使用的前提，横线下方的是目前要证明的结论。接下去的每一行都是一条证明指令（tactic），每条证明指令可以将一个证明目标规约为 0 个，1 个或者更多的证明目标。

```
  intros x.
```

下面的 rewrite 指令可以把待证明结论中的项替换为与一个等价的项。在下面的第一条 rewrite 指令表示将`left_unit`性质的第一个参数代入为`x + (- x)`后进行替换。因此，待证明结论就变为了`0 + (x + (- x)) = 0`。

```
  rewrite <- (left_unit (x + (- x))).
```

下面这条指令中，通过 rewrite 指令后的箭头表明了替换的方向为使用`left_inv`等式的左侧替换该等式的右侧。而指令最后的 at 1 则表示只对第一个可以替换处进行替换。

```
  rewrite <- (left_inv (- x)) at 1.
```

在没有歧义的情况下，并不一定需要填写参数。

```
  rewrite -> assoc.
  rewrite <- (assoc (- x)).
```

如果不使用箭头，则默认为使用向右的箭头。

```
  rewrite left_inv.
  rewrite left_unit.
  rewrite left_inv.
```

最后，reflexivity 指令说的是：等式具有自反性，现在等式两边完全相同，所以已经证明完了。

```
  reflexivity.
Qed.
```

下面可以进一步证明右单位元性质。

```
Theorem right_unit {G: GroupOperator} {GP: GroupProperties G}:
  forall (x: carrier_set), x + 0 = x.
Proof.
  intros.
  rewrite <- (left_inv x).
  rewrite <- assoc.
```

证明中可以使用先前证明的结论。

```
  rewrite right_inv.
  rewrite left_unit.
  reflexivity.
Qed.
```

我们还可以证明双重取逆符号的消去。

```
Theorem inv_involutive {G: GroupOperator} {GP: GroupProperties G}:
  forall (x: carrier_set), - - x = x.
Proof.
  intros.
  rewrite <- (left_unit (- - x)).
  rewrite <- (right_inv x).
  rewrite assoc.
```

在 Coq 中，也可以将多条 rewrite 指令，写到一起。

```
  rewrite right_inv, right_unit.
  reflexivity.
Qed.
```

习题 1.

```
Lemma cancel_PN {G: GroupOperator} {GP: GroupProperties G}:
  forall (x y: carrier_set), x + y + - y = x.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

参考答案.

```
Proof.
  intros.
  rewrite assoc, right_inv, right_unit.
  reflexivity.
Qed.
```

在 Coq 中，`rewrite` 指令除了可以使用通过 `Class` 单独列出的假设以及已经证明的结论还可以使用待证明目标中的前提假设。下面命题说的是，在一个群中，如果 `x + z = y + z` 那么 `x = y`。

```
Lemma cancel_right {G: GroupOperator} {GP: GroupProperties G}:
  forall (x y z: carrier_set),
    x + z = y + z ->
    x = y.
Proof.
  intros.
```

经过 `intros` 之后，`x + z = y + z` 这一前提被命名为 `H`。

```
  rewrite <- (right_unit x), <- (right_unit y).
  rewrite <- (right_inv z).
  rewrite <- ! assoc.
```

下面指令利用前提`H`进行`rewrite`。

```
  rewrite H.
  reflexivity.
Qed.
```

习题 2.

```
Lemma move_right_PN {G: GroupOperator} {GP: GroupProperties G}:
  forall (x y z: carrier_set),
    x + z = y ->
    x = y + - z.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

参考答案.

```
Proof.
  intros.
  rewrite <- H.
  rewrite cancel_PN.
  reflexivity.
Qed.
```