程序语言的语法

1 一个极简的指令式程序语言

```
Module While_SimpleWhile.
```

以下考虑一种极简的程序语言。它的程序表达式分为整数类型表达式与布尔类型表达式，其中整数类型表达式只包含加减乘运算与变量、常数。布尔表达式中只包含整数类型表达式之间的大小比较或者这些比较结果之间的布尔运算。而它的程序语句也只有对变量赋值、顺序执行、if 语句与 while 语句。

整数类型表达式

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

布尔类型表达式

```
    EB :: = TRUE | FALSE | EI < EI | EB && EB | ! EB
```

语句

```
     C :: = SKIP |
            V = EI |
            C; C |
            if (EB) then { C } else { C } |
            while (EB) do { C }
```

下面依次在 Coq 中定义该语言变量名、表达式与语句。

```
Definition var_name: Type := string.
```

```
Inductive expr_int : Type :=
  | EConst (n: Z): expr_int
  | EVar (x: var_name): expr_int
  | EAdd (e1 e2: expr_int): expr_int
  | ESub (e1 e2: expr_int): expr_int
  | EMul (e1 e2: expr_int): expr_int.
```

在 Coq 中，可以利用`Notation`使得这些表达式更加易读。`Notation`的具体定义详见 Coq 源代码。使用`Notation`的效果如下：

```
Check [[1 + "x"]].
Check [["x" * ("a" + "b" + 1)]].
```

```
Inductive expr_bool: Type :=
  | ETrue: expr_bool
  | EFalse: expr_bool
  | ELt (e1 e2: expr_int): expr_bool
  | EAnd (e1 e2: expr_bool): expr_bool
  | ENot (e: expr_bool): expr_bool.
```

```
Inductive com : Type :=
  | CAss (x: var_name) (e: expr_int): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr_bool) (c1 c2: com): com
  | CWhile (e: expr_bool) (c: com): com.



End While_SimpleWhile.
```

2 While 语言

```
Module While_While.
```

在许多以 C 语言为代表的常用程序语言中，往往不区分整数类型表达式与布尔类型表达式，同时表达式中也包含更多运算符。例如，我们可以如下规定一种程序语言的语法。

表达式

```
E :: = N | V | E+E | E-E | E*E | E/E | E%E |
       E<E | E<=E | E==E | E!=E | E>=E | E>E |
       E&&E | E||E | !E
```

语句

```
C :: = SKIP |
       V = E |
       C; C |
       if (E) then { C } else { C } |
       while (E) do { C }
```

下面依次在 Coq 中定义该语言的变量名、表达式与语句。

```
Definition var_name: Type := string.
```

再定义二元运算符和一元运算符。

```
Inductive binop : Type :=
  | OOr | OAnd
  | OLt | OLe | OGt | OGe | OEq | ONe
  | OPlus | OMinus | OMul | ODiv | OMod.

Inductive unop : Type :=
  | ONot | ONeg.
```

下面是表达式的抽象语法树。

```
Inductive expr : Type :=
  | ENum (n: Z): expr
  | EVar (x: var_name): expr
  | EBinop (op: binop) (e1 e2: expr): expr
  | EUnop (op: unop) (e: expr): expr
  | EDeref (e: expr): expr.
```

最后程序语句的定义是类似的。

```
Inductive com : Type :=
  | CAss (x: var_name) (e: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com.



End While_While.
```

3 用 Coq 归纳类型定义二叉树

```
Inductive tree: Type :=
| Leaf: tree
| Node (l: tree) (v: Z) (r: tree): tree.
```

这个定义说的是，一棵二叉树要么是一棵空树`Leaf`，要么有一棵左子树、有一棵右子树外加有一个根节点整数标号。我们可以在 Coq 中写出一些具体的二叉树的例子。

```
Definition tree_example0: tree :=
  Node Leaf 1 Leaf.
```

```
Definition tree_example1: tree :=
  Node (Node Leaf 0 Leaf) 2 Leaf.
```

```
Definition tree_example2a: tree :=
  Node (Node Leaf 8 Leaf) 100 (Node Leaf 9 Leaf).
```

```
Definition tree_example2b: tree :=
  Node (Node Leaf 9 Leaf) 100 (Node Leaf 8 Leaf).
```

```
Definition tree_example3a: tree :=
  Node (Node Leaf 3 Leaf) 5 tree_example2a.
```

```
Definition tree_example3b: tree :=
  Node tree_example2b 5 (Node Leaf 3 Leaf).
```

它们分别表示下面这些树结构

```
          2      100        100
   1     /      /   \      /   \
        0      8     9    9     8
```

```
      5                5
     / \              / \
    3  100          100  3
      / \            / \
     8   9          9   8
```

Coq 中，我们往往可以使用递归函数定义归纳类型元素的性质。Coq 中定义递归函数时使用的关键字是`Fixpoint`。下面两个定义通过递归定义了二叉树的高度和节点个数。

```
Fixpoint tree_height (t: tree): Z :=
  match t with
  | Leaf => 0
  | Node l v r => Z.max (tree_height l) (tree_height r) + 1
  end.
```

```
Fixpoint tree_size (t: tree): Z :=
  match t with
  | Leaf => 0
  | Node l v r => tree_size l + tree_size r + 1
  end.
```

Coq 系统“知道”每一棵特定树的高度和节点个数是多少。下面是具体的 Coq 代码。其中，`Example`关键字与`Theorem`、`Lemma`、`Corollary`的作用是一样的，用于描述一个即将进行证明的性质。而证明中可以直接使用`reflexivity`指令则说明，该指令不仅可以用于等号两侧语法完全相同的情况，其还能基于 Coq 定义（例如此处的递归函数定义）进行化简。

```
Example Leaf_height:
  tree_height Leaf = 0.
Proof. reflexivity. Qed.
```

```
Example tree_example2a_height:
  tree_height tree_example2a = 2.
Proof. reflexivity. Qed.
```

```
Example treeexample3b_size:
  tree_size tree_example3b = 5.
Proof. reflexivity. Qed.
```

Coq 中也可以定义树到树的函数。下面的`tree_reverse`函数把二叉树进行了左右翻转。

```
Fixpoint tree_reverse (t: tree): tree :=
  match t with
  | Leaf => Leaf
  | Node l v r => Node (tree_reverse r) v (tree_reverse l)
  end.
```

下面是三个二叉树左右翻转的例子：

```
Example Leaf_tree_reverse:
  tree_reverse Leaf = Leaf.
Proof. reflexivity. Qed.
```

```
Example tree_example0_tree_reverse:
  tree_reverse tree_example0 = tree_example0.
Proof. reflexivity. Qed.
```

```
Example tree_example3_tree_reverse:
  tree_reverse tree_example3a = tree_example3b.
Proof. reflexivity. Qed.
```

归纳类型有几条基本性质。(1) 归纳定义规定了一种分类方法，以`tree`类型为例，一棵二叉树`t`要么是`Leaf`，要么具有形式`Node l v r`；(2) 以上的分类之间是互斥的，即无论`l`、`v`与`r`取什么值，`Leaf`与`Node l v r`都不会相等；(3) `Node`这样的构造子是函数也是单射。这三条性质对应了 Coq 中的三条证明指令：`destruct`、`discriminate`与`injection`。利用它们就可以证明几条最简单的性质：

```
Lemma Node_inj_left: forall l1 v1 r1 l2 v2 r2,
  Node l1 v1 r1 = Node l2 v2 r2 ->
  l1 = l2.
Proof.
  intros.
  injection H as H_l H_v H_r.
```

上面的`injection`指令使用了`Node`是单射这一性质。

```
  rewrite H_l.
  reflexivity.
Qed.
```

有时，手动为`injection`生成的命题进行命名显得很🅵嗦，Coq 允许用户使用问号占位，从而让 Coq 进行自动命名。

```
Lemma Node_inj_right: forall l1 v1 r1 l2 v2 r2,
  Node l1 v1 r1 = Node l2 v2 r2 ->
  r1 = r2.
Proof.
  intros.
  injection H as ? ? ?.
```

这里，Coq 自动命名的结果是使用了`H`、`H0`与`H1`。下面也使用`apply`指令取代`rewrite`简化后续证明。

```
  apply H1.
Qed.
```

如果不需要用到`injection`生成的左右命题，可以将不需要用到的部分用下划线占位。

```
Lemma Node_inj_value: forall l1 v1 r1 l2 v2 r2,
  Node l1 v1 r1 = Node l2 v2 r2 ->
  v1 = v2.
Proof.
  intros.
  injection H as _ ? _.
  apply H.
Qed.
```

下面引理说：若`Leaf`与`Node l v r`相等，那么`1 = 2`。换言之，`Leaf`与`Node l v r`始终不相等，否则就形成了一个矛盾的前提。

```
Lemma Leaf_Node_conflict: forall l v r,
  Leaf = Node l v r -> 1 = 2.
Proof.
  intros.
  discriminate.
Qed.
```

下面这个简单性质与`tree_reverse`有关。

```
Lemma reverse_result_Leaf: forall t,
  tree_reverse t = Leaf ->
  t = Leaf.
Proof.
  intros.
```

下面的`destruct`指令根据`t`是否为空树进行分类讨论。

```
  destruct t.
```

执行这一条指令之后，Coq 中待证明的证明目标由一条变成了两条，对应两种情况。为了增加 Coq 证明的可读性，我们推荐大家使用 bullet 记号把各个子证明过程分割开来，就像一个一个抽屉或者一个一个文件夹一样。Coq 中可以使用的 bullet 标记有：`+ - * ++ -- **`等等

```
  + reflexivity.
```

第一种情况是`t`是空树的情况。这时，待证明的结论是显然的。

```
  + discriminate H.
```

第二种情况下，其实前提`H`就可以推出矛盾。可以看出，`discriminate`指令也会先根据定义化简，再试图推出矛盾。

```
Qed.
```

4 结构归纳法

我们接下去将证明一些关于`tree_height`，`tree_size`与`tree_reverse`的基本性质。我们在证明中将会使用的主要方法是归纳法。

相信大家都很熟悉自然数集上的数学归纳法。数学归纳法说的是：如果我们要证明某性质`P`对于任意自然数`n`都成立，那么我可以将证明分为如下两步：

- 奠基步骤：证明`P 0`成立；
- 归纳步骤：证明对于任意自然数`n`，如果`P n`成立，那么`P (n + 1)`也成立。

对二叉树的归纳证明与上面的数学归纳法稍有不同。具体而言，如果我们要证明某性质`P`对于一切二叉树`t`都成立，那么我们只需要证明以下两个结论：

- 奠基步骤：证明`P Leaf`成立；
- 归纳步骤：证明对于任意二叉树`l` `r`以及任意整数标签`n`，如果`P l`与`P r`都成立，那么`P (Node l n r)`也成立。

这样的证明方法就成为结构归纳法。在 Coq 中，`induction`指令表示：使用结构归纳法。下面是几个证明的例子。

第一个例子是证明`tree_size`与`tree_reverse`之间的关系。

```
Lemma reverse_size: forall t,
  tree_size (tree_reverse t) = tree_size t.
Proof.
  intros.
  induction t.
```

上面这个指令说的是：对`t`结构归纳。Coq 会自动将原命题规约为两个证明目标，即奠基步骤和归纳步骤。

```
  + simpl.
```

第一个分支是奠基步骤。这个`simpl`指令表示将结论中用到的递归函数根据定义化简。

```
    reflexivity.
  + simpl.
```

第二个分支是归纳步骤。我们看到证明目标中有两个前提`IHt1`以及`IHt2`。在英文中`IH`表示 induction hypothesis 的缩写，也就是归纳假设。在这个证明中`IHt1`与`IHt2`分别是左子树`t1`与右子树`t2`的归纳假设。

```
    rewrite IHt1.
    rewrite IHt2.
    lia.
```

这个`lia`指令的全称是 linear integer arithmatic，可以用来自动证明关于整数的线性不等式。

```
Qed.
```

第二个例子很类似，是证明`tree_height`与`tree_reverse`之间的关系。

```
Lemma reverse_height: forall t,
  tree_height (tree_reverse t) = tree_height t.
Proof.
  intros.
  induction t.
  + simpl.
    reflexivity.
  + simpl.
    rewrite IHt1.
    rewrite IHt2.
    lia.
```

注意：这个`lia`指令也是能够处理`Z.max`与`Z.min`的。

```
Qed.
```

下面我们将通过重写上面这一段证明，介绍 Coq 证明语言的一些其他功能。

```
Lemma reverse_height_attempt2: forall t,
  tree_height (tree_reverse t) = tree_height t.
Proof.
  intros.
  induction t; simpl.
```

在 Coq 证明语言中可以用分号将小的证明指令连接起来形成大的证明指令，其中`tac1 ; tac2`这个证明指令表示先执行指令`tac1`，再对于`tac1`生成的每一个证明目标执行`tac2`。分号是右结合的。

```
  + reflexivity.
  + simpl.
    lia.
```

此处的`lia`指令不仅可以处理结论中的整数线性运算，其自动证明过程中也会使用前提中关于整数线性运算的假设。

```
Qed.
```

习题 1.

```
Lemma reverse_involutive: forall t,
  tree_reverse (tree_reverse t) = t.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

下面证明`tree_reverse`是一个单射。

```
Lemma tree_reverse_inj: forall t1 t2,
  tree_reverse t1 = tree_reverse t2 ->
  t1 = t2.
Proof.
```

这个引理的 Coq 证明需要我们特别关注：真正需要归纳证明的结论是什么？如果选择对`t1`做结构归纳，那么究竟是归纳证明对于任意`t2`上述性质成立，还是归纳证明对于某“特定”的`t2`上述性质成立？如果我们按照之前的 Coq 证明习惯，用 `intros`与`induction t1`两条指令开始证明，那就表示用归纳法证明一条关于“特定”`t2`的性质。

```
  intros.
  induction t1.
  + destruct t2.
```

奠基步骤的证明可以通过对`t2`的分类讨论完成。

```
    - reflexivity.
```

如果`t2`是空树，那么结论是显然的。

```
    - simpl in H.
      discriminate H.
```

如果`t2`是非空树，那么前提`H`就能导出矛盾。如上面指令展示的那样，`simpl`指令也可以对前提中的递归定义化简。当然，在这个证明中，由于之后的`discriminate`指令也会完成自动化简，这条`simpl`指令其实是可以沈略的。

```
  +
Abort.
```

进入归纳步骤的证明时，不难发现，证明已经无法继续进行。因为需要使用的归纳假设并非关于原`t2`值的性质。正确的证明方法是用归纳法证明一条对于一切`t2`的结论。

```
Lemma tree_reverse_inj: forall t1 t2,
  tree_reverse t1 = tree_reverse t2 ->
  t1 = t2.
Proof.
  intros t1.
```

上面这条`intros 1t`指令可以精确地将`t1`放到证明目标的前提中，同时却将`t2`保留在待证明目标的结论中。

```
  induction t1; simpl; intros.
  + destruct t2.
    - reflexivity.
    - discriminate H.
  + destruct t2.
    - discriminate H.
    - injection H as ? ? ?.
      rewrite (IHt1_1 _ H1).
      rewrite (IHt1_2 _ H).
      rewrite H0.
      reflexivity.
Qed.
```