Coq 中的程序语法树

1 一个极简的指令式程序语言

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 :=
  | CSkip: com
  | CAsgn (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.
```

2 While 语言

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

最后程序语句的定义是类似的。

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

3 更多的程序语言：WhileDeref

下面在 While 程序语言中增加取地址上的值`EDeref`操作。

```
Inductive expr : Type :=
  | EConst (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.
```

相应的，赋值语句也可以分为两种情况。

```
Inductive com : Type :=
  | CSkip: com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com.
```

4 更多的程序语言：WhileD

在大多数程序语言中，会同时支持或不支持取地址`EAddrOf`与取地址上的值 `EDeref`两类操作，我们也可以在 WhileDeref 语言中再加入取地址操作。

```
Inductive expr : Type :=
  | EConst (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
  | EAddrOf (e: expr): expr.
```

程序语句的语法树不变。

```
Inductive com : Type :=
  | CSkip: com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com.
```

5 更多的程序语言：WhileDC

下面在程序语句中增加控制流语句 continue 与 break，并增加多种循环语句。

```
Inductive expr : Type :=
  | EConst (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
  | EAddrOf (e: expr): expr.
```

```
Inductive com : Type :=
  | CSkip: com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com
  | CFor (c1: com) (e: expr) (c2: com) (c3: com): com
  | CDoWhile (c: com) (e: expr): com
  | CContinue: com
  | CBreak: com.
```

6 更多的程序语言：WhileDCL

下面在程序语句中增加局部变量声明。

```
C :: = skip | continue | break |
       V = E | * E = E |
       C; C |
       if (E) then { C } else { C } |
       while (E) do { C } |
       for (C; E; E) { C } |
       do { C } while (E)
       local X in { C }
```

```
Inductive com : Type :=
  | CSkip: com
  | CLocalVar (x: var_name) (c1: com): com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com
  | CFor (c1: com) (e: expr) (c2: com) (c3: com): com
  | CDoWhile (c: com) (e: expr): com
  | CContinue: com
  | CBreak: com.
```

7 更多的程序语言：WhileF

下面在程序表达式中增加函数调用，在程序语句中增加过程调用。

```
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 |
      * E | & E | F(E, E, ..., E)
```

```
C ::= skip | continue | break | return
      V = E | * E = E |
      P(E, E, ..., E) |
      C; C |
      if (E) then { C } else { C } |
      while (E) do { C } |
      for (C; E; E) { C } |
      do { C } while (E)
      local X in { C }
```

除了在程序表达式和程序语句中增加新的语法结构之外，程序中还要新增一类新对象：函数与过程。

- 函数 f
  - 函数名：`f.(name_of_func)`
  - 函数参数变量列表：`f.(args_of_func)`
  - 函数体：`f.(body_of_func)`
- 过程 p
  - 过程名：`p.(name_of_proc)`
  - 过程参数变量列表：`p.(args_of_proc)`
  - 过程体：`p.(body_of_proc)`

```
Definition func_name: Type := string.
Definition proc_name: Type := string.
```

下面是新的表达式定义，这是一个嵌套递归定义。

```
Inductive expr : Type :=
  | EConst (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
  | EAddrOf (e: expr): expr
  | EFuncCall (f: func_name) (es: list expr).
```

下面是新的程序语句定义，这也是一个嵌套递归定义。这里约定`return_var`是一个特定的表示返回值的变量，而返回指令本身没有参数。

```
Definition return_var: var_name := "__return".
```

```
Inductive com : Type :=
  | CSkip: com
  | CLocalVar (x: var_name) (c1: com): com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CProcCall (p: proc_name) (es: list expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com
  | CFor (c1: com) (e: expr) (c2: com) (c3: com): com
  | CDoWhile (c: com) (e: expr): com
  | CContinue: com
  | CBreak: com
  | CReturn: com.
```

下面定义程序中的函数与过程。

```
Record func: Type := {
  name_of_func: func_name;
  body_of_func: com;
  args_of_func: list var_name;
}.
```

```
Record proc: Type := {
  name_of_proc: proc_name;
  body_of_proc: com;
  args_of_proc: list var_name;
}.
```

最后，一段完整的程序由全局变量列表、函数列表、过程列表与入口函数组成。

```
Record prog: Type := {
  gvars: list var_name;
  funcs: list func;
  procs: list proc;
  entry: func_name
}.
```

8 简单语法变换与证明

习题 1. 下面的递归函数`remove_skip`定义了在程序语句中删除多余空语句的操作。

```
Fixpoint remove_skip (c: com): com :=
  match c with
  | CSeq c1 c2 =>
      match remove_skip c1, remove_skip c2 with
      | CSkip, _ => remove_skip c2
      | _, CSkip => remove_skip c1
      | _, _ => CSeq (remove_skip c1) (remove_skip c2)
      end
  | CIf e c1 c2 =>
      CIf e (remove_skip c1) (remove_skip c2)
  | CWhile e c1 =>
      CWhile e (remove_skip c1)
  | _ =>
      c
  end.
```

下面请证明：`remove_skip`之后，确实不再有多余的空语句了。所谓没有空语句，是指不出现`c; SKIP`或`SKIP; c`这样的语句。首先定义：局部不存在多余的空语句。

```
Definition not_sequencing_skip (c: com): Prop :=
  match c with
  | CSeq CSkip _ => False
  | CSeq _ CSkip => False
  | _ => True
  end.
```

其次定义语法树的所有子树中都不存在多余的空语句。

```
Fixpoint no_sequenced_skip (c: com): Prop :=
  match c with
  | CSeq c1 c2 =>
      not_sequencing_skip c /\
      no_sequenced_skip c1 /\ no_sequenced_skip c2
  | CIf e c1 c2 =>
      no_sequenced_skip c1 /\ no_sequenced_skip c2
  | CWhile e c1 =>
      no_sequenced_skip c1
  | _ =>
      True
  end.
```

下面是需要证明的结论。

```
Theorem remove_skip_no_sequenced_skip: forall c,
  no_sequenced_skip (remove_skip c).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```