An Axiomatic Definition of the Programming Language PASCAL

C.A.R. Hoare

0 Introduction

The purpose of a formal language definition is to act as a "contract" between the implementor and user of the language. It is not intended to be read straight through, but rather to be used by both parties as a work of reference; it will also be used as the basis of more readable descriptions of the language. Thus the objective of a formal description is to make it easy to identify the chapter and subsection relevant to any query, and to consult it in isolation, without having to follow cross-references to other sections and chapters. The axiomatic definition of PASCAL seems to achieve this objective to a remarkable degree.

The language treated in this paper is a version of PASCAL which has been simplified in several respects.

(1) No variable occuring in the record variable part of a with statement may be changed within its body.

(2) A procedure may not access non-local variables other than those in its parameter list.

(3) All actual parameters corresponding to formals specified as var must be distinct identifiers.

There are other changes to PASCAL which would considerably simplify the axiomatisation, and might also be of benefit to its user.

(1) The alfa type could be regarded as just a packed representation for a character array.

(2) Conditional and case expressions could be allowed.

(3) Explicit constructors and type transfer functions for defined types could be included in the language.

(4) Some simplification of the file type is to be recommended.

(5) The variable parameters of a procedure should be indicated on the call side as well as on declaration.

(6) Exclusion of backward jumps and mutual recursion would simplify the proof rules.

(7) The replacement of classes by partial mappings and permitting recursive data type definitions would make the language into a more high-level tool.

But there are very good reasons for avoiding these "improvements", since their inclusion would undoubtedly detract from the very high qual ity of the PASCAL implementation. In particular, point (7) would entirely change the nature of the language.

The treatment given in this paper is not wholly formal, and depends on the good will of the reader. In particular

(1) Free variables in axioms are assumed to be universally quantified.

(2) The expression of the "induction" axiom is left informal.

(3) The types of the variables used may be supplied by context.

(4) The name of a type is used as a transfer function constructing a value of the type. Such use of the type name is not available to the programmer.

(5) Axioms for a defined type must be modelled after the definition, and be applied only in the block to which the definition is local.

(6) The context-free and context-dependent aspects of the syntax are defined in [2], and are not repeated here.

(7) Where hypotheses and subsidiary deductions are given in the proof rules, the hypotheses may be accumulated, and the subsidiary deductions may be nested in the usual manner.

(8) The use of a type name in its own definition is not permitted to the PASCAL programmer.

The following abbreviations are standard:

(1) $n < m$ for $\neg m \leq n$
$n \geq m$ for $m \leq n$
$n > m$ for $m < n$

(2) $s_1, s_2, \ldots, s_n : T$ for $s_1 : T; s_2 : T; \ldots; s_n : T$

(3) $P\{Q\}R$, where P and R are propositional formulae and Q is a statement of PASCAL is interpreted as stating that if P is true of the program variables before initiating execution of Q, then R will be true on termination. If Q never terminates, or exits by a jump to a label outside itself, $P\{Q\}R$ is vacuously true.

(4) X_y^x stands for the result of substituting y for all the free occurrences of x in X. If y contains free variables which would thereby become bound, this unintended collision is avoided by preliminary change in the name of the bound variables of X.

The axioms for data and data structures comprise about half the definition. They are modelled on the constructive definition of integers. Axioms defining the basic operators are modelled on primitive recursive definitions, so that it is easy to check that they define a unique computable result for every desired combination of operands, and leave other cases (e.g. division by zero) undefined. The axioms are formulated to apply to the normal unbounded domains of mathematics. An implementor is free to discontinue execution of any program which invokes operations which would exceed the range of efficient operation of his machine (e.g. integer overflow, or exhaustion of stack space). However, to continue execution with the wrong result would violate this axiomatic definition.

Most of the formal material of this paper is more fully explained in the references. The main extensions made by this paper are:

(1) Definitions of files and classes

(2) Treatment of mutual recursion

(3) Attempted definition of the whole of a large, efficient, and useful language.

1 Standard Scalar Types

1.1 The Integer Type.

1.1.1 0 is an integer.

1.1.2 if n is an integer, so are succ(n) and pred(n)

1.1.3 These are the only integers.

1.1.4 $n = succ(pred(n)) = pred(succ(n))$.

1.1.5 $n \leq n$.

1.1.6 $n < succ(n)$.

1.1.7 $n \leq m \supset n \leq succ(m)$.

1.1.8 $m < n \supset m < succ(n)$.

1.1.9 $n + 0 = n$.

1.1.10 $n + m = pred(n) + succ(m) = succ(n) + pred(m)$.

1.1.11 $n - 0 = n$.

1.1.12 $n - m = succ(n) - succ(m) = pred(n) - pred(m)$.

1.1.13 $n * 0 = 0$

1.1.14 $n * m = n * succ(m) - n = n * pred(m) + n.$

1.1.15 $n > 0 \supset m - n < (m \underline{div}\ n) * n \leq m.$

1.1.16 $n < 0 \supset m \leq (m \underline{div}\ n) * m < m - n.$

1.1.17 $m \underline{mod}\ n = m - ((m \underline{div}\ n) * n).$

1.1.18 $abs(m) = \underline{if}\ m < 0\ \underline{then}\ -m\ \underline{else}\ m.$

1.1.19 $sqr(m) = m * m.$

1.1.20 $odd(m) = (m \underline{mod}\ 2 = 1).$

1.1.21 an integer constant consisting of digits

$$d_n d_{n-1} \ldots d_0 \text{ means}$$
$$10^n * d_n + 10^{n-1} * d_{n-1} + \ldots + 10^0 * d_0$$

1.1.22 1 means succ(0), 2 means succ(1), ..., 9 means succ(8)

1.2 The Boolean Type.

1.2.1 false and true are distinct Boolean values.

1.2.2 These are the only Boolean values.

1.2.3 true = succ(false) & false = pred(true)

1.2.4 $false \leq x \leq true.$

1.2.5 $\neg(true \leq false).$

1.2.6 $(\neg false) = true \ \&\ (\neg true) = false.$

1.2.7 $true \wedge true = true$

1.2.8 $false \wedge x = x \wedge false = false$

1.2.9 $false \vee false = false$

1.2.10 $true \vee x = x \vee true = true.$

1.3 The Character Type.

1.3.1 "A", "B", ..., are distinct elements of type Char.

1.3.2 These are the only elements of type Char.

1.3.3 if x is a Char, then int(x) is an integer and x = chr(int(x)).

1.3.4 if n is an integer in the range of int, then chr(n) is a character and n = int(chr(n)).

1.3.5 if x and y are Char then

$$x \leq y \equiv int(x) \leq int(y).$$

(1) The set of constants denoting values of type Char may be selected by the implementor. They should begin and end with the symbol chosen by the implementor to denote quotation.

1.4 The Alfa Type.

1.4.1 the type name "alfa" is an abbreviation of array [1..alfalength] of Char.

1.4.2 $x \prec y \equiv$
$\exists n\ (1 \leq n \leq \text{alfalength} \ \&\ x[n] \prec y[n]$
$\&\ \forall m(1 \leq m \prec n \supset x[m] = y[m]))$.

1.4.3 unpack(z,a,i) means
for j = 1 to alfalength do a[i + j - 1] : = z[j].

1.4.4 pack(a,i,z) means
for j = 1 to alfalength do z[j] : = a[i + j - 1].

1.4.5 If $c_1, c_2, \ldots, c_n$ are characters and $n \leq$ alfalength and
c = "$c_1c_2\ldots c_n$" then
c[1] = "c_1", c[2] = "c_2",..., c[alfalength] = "$c_{\text{alfalength}}$".
where if n < alfalength, $c_{n+1},\ldots, c_{\text{alfalength}}$ are all equal to the character blank.

(1) For the properties of array types see 2.3.

(2) The programmer is not permitted to subscript a variable declared as alfa.

(3) The constants alfalength and blank are not available to the programmer.

1.5 The Real Type.

I dont know of any wholly successful axiomatisation of the real type (floating point). The ideal solution would be, as in the case of integers, to give axioms for the real continuum familiar to mathematicians; and certainly this will considerably simplify the task of program proving. Unfortunately, the properties of floating point numbers are very different from those of true reals; and (in contrast to the case of integers), it is not practical merely to permit an implementation to refuse to execute programs which invoke operations involving loss of significance

Thus it appears necessary to construct a separate machine-independent axiomatisation for floating point arithmetic. The axiomatisation must describe the relationship between floating point and true reals; and the use of axioms in practical program proofs is likely to be at least as complicated as the error analysis techniques practised by numerical mathematicians.

2 Defined Types

2.1 Scalar Types.

$T = (k_1, k_2, \ldots, k_n)$;

2.1.1 $k_1, k_2, \ldots, k_n$ are distinct elements of T.
2.1.2 These are the only elements of T.
2.1.3 $k_2 = succ(k_1)$ & ... & $k_n = succ(k_{n-1})$.
2.1.4 $k_1 = pred(k_2)$ & ... & $k_{n-1} = pred(k_n)$.
2.1.5 $k_1 \leq x \leq k_n$.
2.1.6 If $y \neq k_n$ then $x \leq y \equiv succ(x) \leq succ(y)$.
2.1.7 case x of $(k_1:y_1, k_2:y_2, \ldots, k_n:y_n)$
= if $x = k_1$ then y_1
else if $x = k_2$ then y_1
.....
else if $x = k_{n-1}$ then y_{n-1}
else y_n.
2.1.8 $k_i:k_j:\ldots k_m:y$ (in a case construction) means
$k_i:y, k_j:y, \ldots, k_m:y$.

(1) The case expression defined in 2.1.7 is not available to the programmer.

2.2 Subrange Types.

type T = min..max;
where min and max are of type T_0

Let a, b $\in T_0$ and $min \leq a \leq b \leq max$, and x, y $\in$ T.

2.2.1 If $a \in T_0$ and $min \leq a \leq max$ then T(a) is a T.
2.2.2 These are the only elements of T.
2.2.3 $T^{-1}(T(a)) = a$.
2.2.4 If θ is any monadic operator defined on T_0 then
θx means $\theta(T^{-1}(x))$
2.2.5 If $\oplus$ is any binary operator defined on type T_0, then
$x \odot y$ means $T^{-1}(x) \oplus T^{-1}(y)$
$x \odot a$ means $T^{-1}(x) \oplus a$
$a \odot x$ means $a \oplus T^{-1}(x)$
2.2.6 x := a means x := T(a)
a := x means a := $T^{-1}(x)$.

(1) Any operation on subtype operands causes automatic conversion the base type. This was not made explicit in [2].

2.3 Array Types.

type T = array D of R;

2.3.1 If r is an R then T(r) is a T.

2.3.2 If d is a D and r is an R and a is a T (a,d:r) is a T.

2.3.3 These are the only elements of T.

2.3.4 $(T(r))[d] = r$.

2.3.5 $(a,d:r)[d'] = \underline{\text{if}}\ d = d'\ \underline{\text{then}}\ r\ \underline{\text{else}}\ a[d']$.

2.3.6 $a = a' = \forall d\ (a[d] = a'[d])$.

2.3.7 $a[d] := r$ means $a := (a,d:r)$.

2.3.8 $\underline{\text{array}}\ [D_1,\ldots,D_n]\ \underline{\text{of}}\ R$ means $\underline{\text{array}}\ [D_1]$ of $(\underline{\text{array}}\ [D_2,\ldots,D_n]\ \underline{\text{of}}\ R)$.

2.3.9 $a[d_1,d_2,\ldots,d_n]$ means $(a[d_1])[d_2,\ldots,d_n]$.

(1) T(r) is the constant array, all of whose component values are equal to r.

(2) (a,d:r) is the result of assigning the value r to element d in the array a.

(3) Neither of these operations is explicitly available in PASCAL.

2.4 Record Types.

type T = record $s_1:T_1$; $s_2:T_2$;...;$s_n:T_n$ end;

2.4.1 If x_1 is a T_1, x_2 is a T_2, x_n is a T_n then $T(x_1,x_2,\ldots,x_n)$ is a T

2.4.2 These are the only elements of T.

2.4.3 $T(x_1,x_2,\ldots,x_n).s_1 = x_1$.
$T(x_1,x_2,\ldots,x_n).s_2 = x_2$.
.....
$T(x_1,x_2,\ldots,x_n).s_n = x_n$.

2.4.4 $x.s_1 := x_1$ means $x := T(x_1,x.s_2,\ldots,x.s_n)$.
$x.s_2 := x_2$ means $x := T(x.s_1,x_2,\ldots,x_n.s_n)$.
.....
$x.s_n := x_n$ means $x := T(x.s_1,x.s_2,\ldots,x_n)$.

2.5 Union Types.

$\underline{\text{type}}$ T = $\underline{\text{case}}$ d:D $\underline{\text{of}}$ k_1:(s_{11}:T_{11};s_{12}:T_{12};...;s_{1n_1}:T_{1n_1});
k_2:(s_{21}:T_{21};s_{22}:T_{22};...;s_{2n_2}:T_{2n_2});
.....;
k_m:(s_{m1}:T_{m1};s_{m2}:T_{m2};...;s_{mn_m}:T_{mn_m});

Let $x_{ij} \in T_{ij}$ for $i \in 1..m$, $j \in 1..n$

2.5.1 The following are distinct elements of T:
$T.k_1(x_{11},x_{12},...,x_{1n_1})$
$T.k_2(x_{21},x_{22},...,x_{2n_2})$
.....
$T.k_m(x_{m1},x_{m2},...,x_{mn_m})$.

2.5.2 These are the only elements of T.

2.5.3 $(T.k_i(x_{i1},x_{i2},...,x_{in_i})).d = k_i$ for $i \in 1..m$.

2.5.4 $(T.k_i(x_{i1},x_{i2},...,x_{in_i})).s_{ij} = x_{ij}$ for $i \in 1..m$, $j \in 1..n_i$.

(1) In PASCAL, a union type may feature only as the type of the last component in a record type; and this component has no selector.

(2) The use of i and j and indices in these axioms is for purposes of abbreviation only. For any given union type, the relevant axioms may easily be written out in full.

2.6 Powerset Types.

$\underline{\text{type}}$ T = $\underline{\text{powerset}}$ T_0;

Let $x_0, y_0 \in T_0$.

2.6.1 [ ] is a T.

2.6.2 If x is a T and x_0 is a T_0,
then $x \vee [x_0]$ is a T.

2.6.3 These are the only elements of T.

2.6.4 $\neg x_0$ $\underline{\text{in}}$ [ ].

2.6.5 x_0 $\underline{\text{in}}$ $(x \vee [x_0])$.

2.6.6 $x_0 \neq y_0 \supset (x_0 \underline{\text{in}} (x \vee [y_0]) \equiv x_0 \underline{\text{in}} x)$.

2.6.7 $x = y \equiv \forall x_0 (x_0 \underline{\text{in}} x \equiv x_0 \underline{\text{in}} y)$.

2.6.8 $x_0 \underline{\text{in}} (x \vee y) \equiv (x_0 \underline{\text{in}} x)\ (x_0 \underline{\text{in}} y)$.

2.6.9 $x_0 \underline{\text{in}} (x \wedge y) = (x_0 \underline{\text{in}} x) \& (x_0 \underline{\text{in}} y)$.

2.6.10 $x_0 \underline{\text{in}} (x - y) = (x_0 \underline{\text{in}} x)\ \& \neg (x_0 \underline{\text{in}} y)$.

2.6.11 $[x_1,x_2,...,x_n]$ means $(...(([\,] \vee [x_1]) \vee [x_2]) \vee ...) \vee [x_n]$.

(1) $[\,]$ is the empty set, $\lceil x_0 \rfloor$ is the unit set of x_0.

(2) This theory of hierarchically typed finite sets suffers from none of the traditional problems of set theory.

2.7 Sequence Types.

type T = sequence T_0;

Let $x_0, x_0' \in T_0$.

2.7.1 $\langle\!\langle\,\rangle\!\rangle$ is a T.

2.7.2 If s is a T and x_0 is a T_0 then $s \frown \langle\!\langle x_0 \rangle\!\rangle$ is a T

2.7.3 These are the only elements of T

2.7.4 $s \frown \langle\!\langle x_0 \rangle\!\rangle \neq \langle\!\langle\,\rangle\!\rangle$

2.7.5 $s \frown \langle\!\langle x_0 \rangle\!\rangle = s' \frown \langle\!\langle x_0' \rangle\!\rangle \equiv s = s' \;\&\; x_0 = x_0'$

2.7.6 $first(\langle\!\langle\,\rangle\!\rangle \frown \langle\!\langle x_0 \rangle\!\rangle) = x_0$

$s \neq \langle\!\langle\,\rangle\!\rangle \supset first(s \frown \langle\!\langle x_0 \rangle\!\rangle) = first(s)$

2.7.7 $tail(\langle\!\langle\,\rangle\!\rangle \frown \langle\!\langle x_0 \rangle\!\rangle) = \langle\!\langle\,\rangle\!\rangle$

$tail(s \frown \langle\!\langle x_0 \rangle\!\rangle) = tail(s) \frown \langle\!\langle x_0 \rangle\!\rangle$

2.7.8 $\langle\!\langle x_1, x_2, \ldots, x_n \rangle\!\rangle$ means $(\ldots((\langle\!\langle\,\rangle\!\rangle \frown \langle\!\langle x_1 \rangle\!\rangle) \frown \langle\!\langle x_2 \rangle\!\rangle) \frown \ldots) \frown \langle\!\langle x_n \rangle\!\rangle$

(1) Sequences are not represented directly in PASCAL; they are required in the definition of files.

(2) $\langle\!\langle\,\rangle\!\rangle$ is the empty sequence, $\langle\!\langle x_0 \rangle\!\rangle$ the sequence whose only element is x_0, and $\frown$ is the operator for concatenation.

(3) first picks out the first element of a non-empty sequence.

(4) the tail is obtained by striking off from a non-empty sequence its first element.

2.8 File Types.

2.8.1 type T = file of D means:

```
type T' = record all, rest: sequence D;
                      this: D;
                       eof: Boolean;
                      mode: (in, out, neutral)
          end
```

2.8.2 the declaration f:T means

f:T; f:=T'($\langle\!\langle\,\rangle\!\rangle$, $\langle\!\langle\,\rangle\!\rangle$, undefined, false, neutral),

where the assignment is considered as moved to the right

of any following declarations.

2.8.3 put(f) means:

```
with f do
    begin if mode = in then go to error;
          if mode = neutral then
             begin mode := out;
                   all := ⟦ ⟧;
                  rest := ⟦ ⟧;
             end;
          all := all↑⟦this⟧
    end
```

2.8.4 reset(f) means

```
with f do
       begin rest := all;
             mode := neutral;
              eof := false;
             this := undefined
       end;
```

2.8.5 get(f) means

```
with f do
       begin if eof ∨ mode = out then go to error;
             if mode = neutral then mode := in
             else if rest = empty then eof := true
             else begin this := first(rest);
                        rest := tail(rest)
                  end;
       end;
```

2.8.6 f↑ means f.this

(1) all contains the value of the whole file; and rest contains that part of the file which remains to be read (empty in the case of an output file). this contains the value of the current item.

2.9 Partial Mapping Types.

```
type T = D ⇸R;
```

2.9.1 omega is a T.

2.9.2 if t is a T, d is a D and r is a R
then (t,d:r) is a T.

2.9.3 These are the only elements of T.

2.9.4 (t,d:r)⟦d'⟧ = if d = d' then r else t⟦d'⟧.

2.9.5 domain(omega) = [ ].
2.9.6 domain(t,d:r) = domain(t)∨[d].
2.9.7 t = t' ≡ (domain(t) = domain(t') &
∀d(d in domain(t)⊃t[d] = t[d'])).

(1) Partial mappings do not feature directly in PASCAL; they are required in the definition of classes.

(2) omega is the partial mapping that is everywhere undefined. domain(t) is the set of subscripts d for which t[d] is defined.

2.10 Classes and Pointer Types.

c:class of R means
type C = ((↑c) ⇒ R); var c:C;c := omega;
where C is a new type name exclusive to this purpose and the assignment is considered as moved to the right of any subsequent declarations.

2.10.1 nil is a ↑c.
2.10.2 next(c) is a ↑c.
2.10.3 These are the only elements of c.
2.10.4 ¬nil in domain(c).
2.10.5 ¬next(c) in domain(c).
2.10.6 if p is a ↑c, then p↑ means c[p]
2.10.7 alloc(p) means begin p := next(c);
if p ≠ nil then c := (c,p:arbitrary) end
2.10.8 alloc(p,t) means the same as alloc(p).

(1) next(c) yields the pointer value of the next "free" location in the class; and nil if the class is full.

(2) This axiomatisation deals with only one variable declared as of class type. Where several variables are declared to be of the same class type, this should be regarded merely as an abbreviation for a program containing separate class declarations for each such variable.

(3) The axioms are deliberately incomplete; it is not known whether they are sufficiently complete for proof purposes.

(4) The declaration of a maximum number of elements in a class is not represented in these axioms; it seema to belong more to the implementation strategy than to the language itself.

3 Statements

3.1 Assignement Statements.

$$R^x_e \ \{x:=e\}R$$

3.2 Composition.

$$\frac{P\ \{Q_1\}S_1,\ S_1\ \{Q_2\}S_2,\ldots,S_{n-1}\ \{Q_n\}R}{P\ \{Q_1;\ Q_2;\ \ldots,\ ;\ Q_n\}R}$$

3.3 Conditional Statements.

3.3.1
$$\frac{P_1\ \{Q_1\}R,\ P_2\ \{Q_2\}R}{(\underline{\text{if}}\ B\ \underline{\text{then}}\ P_1\ \underline{\text{else}}\ P_2)\ \{\underline{\text{if}}\ B\ \underline{\text{then}}\ Q_1\ \underline{\text{else}}\ Q_2\}R}$$

$$\frac{P\ \{Q\}R}{(\underline{\text{if}}\ B\ \underline{\text{then}}\ P\ \underline{\text{else}}\ R)\ \{\underline{\text{if}}\ B\ \underline{\text{then}}\ Q\}R}$$

3.3.2
$$\frac{P_1\ \{Q_1\}R,\ P_2\ \{Q_2\}R,\ldots,P_n\ \{Q_n\}R}{\underline{\text{case}}\ e\ \underline{\text{of}}\ (k_1:P_1,k_2:P_2,\ldots,k_n:P_n)\ \{\underline{\text{case}}\ e\ \underline{\text{of}}\ k_1:Q_1;\ k_2:Q_2;\ldots;k_n:Q_n\underline{\text{end}}\}R}$$

(1) $k_1,k_2,\ldots,k_n$ stand for sequences of one or more constants (separated by :) of the same type as e.

3.4 Repetitive Statements.

3.4.1
$$\frac{S\ \{Q\}P,\ P \supset \underline{\text{if}}\ B\ \underline{\text{then}}\ S\ \underline{\text{else}}\ R}{P\ \{\underline{\text{while}}\ B\ \underline{\text{do}}\ Q\}R}$$

3.4.2
$$\frac{S\ \{Q\}P,\ P \supset \underline{\text{if}}\ B\ \underline{\text{then}}\ R\ \underline{\text{else}}\ S}{S\ \{\underline{\text{repeat}}\ Q\ \underline{\text{until}}\ B\}R}$$

3.4.3 $\underline{\text{for}}\ v:=e1\ \underline{\text{to}}\ e2\ \underline{\text{do}}\ Q$ means

```
exhausted:=(e1 > e2); if ¬ exhausted then v:=e1;
while ¬ exhausted do begin Q; if v < e2 then v:=succ(v)
                                        else exhausted:=true
                                    end
```

where exhausted is a Boolean variable specific to this for statement and local to the smallest procedure containing it.

3.4.4 for v:=e1 downto e2 do Q means
exhausted:=(e2 > e1);
if ¬ exhausted then v:=e1;
while ¬ exhausted do begin Q; if v > e2 then v:=pred(v)
else exhausted:=false
end

(1) The body of a for loop may not change the value of the counting variable v, nor the value of the limit e2.

3.5 Procedure Statements.

$$R \begin{matrix} a_1, a_2, \ldots, a_m \\ f_1(\underline{a}), f_2(\underline{a}), \ldots, f_m(\underline{a}) \end{matrix} \qquad \{f(a)\}R$$

(1) $\underline{a}$ is the list of actual parameters. $a_1, a_2, \ldots, a_m$ are those actual parameters corresponding to formal parameters specified as var. $f_1, f_2, \ldots, f_m$ are specific to and stand for functions yielding the values left in $a_1, a_2, \ldots, a_m$ by an application of the procedure f to a.

3.6 Labels and go to Statements.

3.6.1
$$\frac{S_1\{\text{go to } \ell_1\}\ \text{false}, \ldots, S_n\{\text{go to } \ell_n\}\ \text{false} \quad \vdash P\{Q_1\}S_1, S_1\{Q_2\}S_2, \ldots, S_{n-1}\{Q_n\}R}{P\{\text{begin } Q_1; \ell_1: Q_2; \ldots; {}_n: Q_n \text{ end}\}R}$$

3.6.2
$$\frac{S\{Q\}R,\ P \supset S}{P\{Q\}R}$$

3.6.3
$$\frac{P\{Q\}S,\ S \supset P}{P\{Q\}R}$$

(1) In this rule, the Q_i stand for sequences of none or more statements separated by semicolons.

(2) This rule is very similar to the rule of composition (3.2) expect that in the proof of the component statements, certain assumptions

may be made, which specify the properties of any jumps which they contain.

3.7 With Statements.

$$\frac{P\{Q\}R}{\underline{\text{with}}\ r\ \underline{\text{take}}\ P\ \ \underline{\text{with}}\ r\ \underline{\text{do}}\ Q\ \ \underline{\text{with}}\ r\ \underline{\text{take}}\ R}$$

(1) Neither r, nor any variable occuring in r, (for example as a subscript) may be changed by Q.

(2) with r take P means the same as the result of replacing in P every free occurrence of a selector s defined for r by r.s (i.e. the result of performing on P the same substitutions as PASCAL specifies for Q).

4 Declarations

4.1 Constant Definitions.

$\underline{\text{const}}\ x_1 = k_1,\ x_2 = k_2,\ldots,x_n = k_n;Q$ means

$$Q\begin{matrix}x_1,x_2,\ldots,x_n\\ k_1,k_2,\ldots,k_n\end{matrix}$$

(1) Q is of the form:
< type definition part > < variable declaration part >
< procedure and functions declaration part > < statement part >

4.2 Variable Declarations.

$$\frac{P\{Q^x_y\}R}{P\{x:T;Q\}R}$$

(1) y does not occur free in P,Q or R.

(2) Q takes the form:
{ < variable declarations >;}* < procedure and functions declaration part > < statement part >

(3) The first variable declaration in any procedure is preceded by var, which is ignored for proof purposes.

4.3 Procedure and Function Declarations.

4.3.1

$$\frac{P_1\{h_1\}R_1, P_2\{h_2\}R_2, \ldots, P_n\{h_n\}R_n \vdash P_1\{Q_1\}R_1, P_2\{Q_2\}R_2, \ldots, P_n\{Q_n\}R_n, P\{Q\}R}{P\{h_1;Q_1;h_2;Q_2;\ldots;h_n;Q_n;Q\}R}$$

4.3.2 $P\{\underline{\text{functions}}\ f(\underset{\sim}{x})\}R \vdash \forall_{\underset{\sim}{z}}\ (P \supset R^{f}_{f(x)})$

4.3.3 $P\{\underline{\text{procedure}}\ f(\underset{\sim}{x})\}R \vdash \forall_{\underset{\sim}{z}}\ (P \supset R^{x_1,x_2,\ldots,x_m}_{f_1(\underset{\sim}{x}),f_2(\underset{\sim}{x}),\ldots,f_m(\underset{\sim}{x})})$

(1) $h_1, h_2, \ldots, h_n$ are procedure or function headings; $Q_1, Q_2, \ldots, Q_n$ are the corresponding procedure bodies, each consisting of < constant definition part > < type definition part > < variable declaration part > < procedure and function declaration part > < statement part >. Q is the statement part of the whole construction.

(2) $\underset{\sim}{x}$ is the list of formal parameters; $x_1, x_2, \ldots, x_m$ is the list of those formal parameters as var. $\underset{\sim}{z}$ is a list of all free identifiers of the formula which follows is, excluding the function names $f, f_1, f_2, \ldots, f_m$, and any other free function names of $Q_1, Q_2, \ldots, Q_n$.

(3) $f_1, f_2, \ldots, f_m$ act as functions yielding the values left by the procedure body in its variable parameters $x_1, x_2, \ldots, x_m$.

(4) The use of the procedure heading in the hypotheses is a formal trick to simplify the combination of proof rules for functions and procedures.

Acknowledgement

The author gratefully acknowledges his indebtedness to N. Wirth for many discussions on PASCAL and the proof rules appropriate for it.

References

[1] Floyd, R.W. - Assigning Meanings to Programs. Proc. Amer. Math. Soc. Symposium in Applied Mathematics. Vol. 19, 19-31.

[2] Wirth, N. - The Programming Language PASCAL. Acta Informatica I 1. (1971), 35-63.

[3] Hoare, C.A.R. - An Axiomatic Approach to Computer Programming. Commun. ACM. 12, 10 (October 1969), 576-580, 583.

[4] Hoare, C.A.R. - Procedures and Parameters; an Axiomatic Approach. Symposium on Semantics of Algorithmic Languages. Ed. E. Engeler, Springer-Verlag, 1970. 102-115.

[5] Clint, M.& Hoare, C.A.R. - Jumps and Functions: an Axiomatic Approach Acta Informatica (to appear).

[6] Hoare, C.A.R. - Notes on Data Structuring (to appear).