I. INTRODUCTION

The present paper is a progress report about our work on semantics and proof theory of programming languages. We study a number of fundamental programming concepts occurring e.g. in the language PASCAL, viz. assignment, sequential composition, conditionals, locality, and (recursive) procedures with parameters called-by-value and called-by-variable. Our goal is the development of a formalism which satisfies two requirements:
- Semantic adequacy: the definitions capture exactly the meaning attributed to these concepts in the PASCAL report.
- Mathematical adequacy: The definitions are as precise and mathematically rigorous as possible.

Of course, full semantic adequacy cannot be achieved within the scope of our paper. Thus, we were forced to omit certain aspects of the concepts concerned. What we hope to have avoided, however, is any essential alteration of a concept for the sake of making it more amenable to formal treatment.

Our approach follows the method of denotational semantics introduced by Scott and Strachey (e.g. in [12]). Moreover, we investigate the connections between denotational semantics and Hoare's proof theory ([6]), insofar as pertaining to the concepts mentioned above.

As main contributions of our paper we see:
- The proposal of a new definition of substitution for a subscripted variable. This allows an extension of Hoare's axiom for assignment to the case of assignment to a subscripted variable. (This idea is described in greater detail in [2].)
- The proposal of a semantic definition and corresponding proof rule for recursive procedures with an adequate treatment of call-by-value and call-by-variable. (We believe these to be new. The proof rule is based on Scott's (or computational) induction, which is well-understood for parameterless procedures, but hardly so for procedures with parameters. In our opinion, neither the papers of Manna et al. (e.g. in [10,11]) nor those of e.g. De Bakker ([1]), Hoare ([7]), Hoare and Wirth ([8]), Igarashi, London and Luckham ([9]) give the full story on this subject.)

It will turn out that our treatment of procedures is quite complex. However, we doubt whether an approach which is essentially simpler is possible. Of course, we do not claim that our formalism is the last word, but the programming notions involved are intricate,
and we feel that essential simplification could be obtained only by changing the lan-
guage.

The paper has the following outline:
Section 2 gives the syntax of the various language constructs. Also, a careful defini-
tion of substitution is given which is needed for the treatment of assignment, local-
ity and parameter passing.
Section 3 is devoted to the definition of the denotational semantics of the five types
of statements. We introduce the semantic function \( M \) which gives meaning to a statement
\( S \), in a given environment \( e \) (a mapping from variables to addresses) and store \( s \) (a mapp-
ing from addresses to values), yielding a new store \( \sigma' : M(S)(e,\sigma) = \sigma' \). For assign-
ment, sequential composition and conditionals the definitions are fairly straightforward.
It is also reasonably clear what to do about locality, but the treatment of pro-
cedures may be rather hard to follow. Some of the causes are:
- When applying the usual least fixed point approach, one has to be careful with the
types (in the set-theoretical sense) of the functions involved.
- The notion of call-by-variable (the FORTRAN call-by-reference) requires a somewhat
mixed action to be taken: When the actual parameter (which has to be a variable) is
subscripted, the subscript is evaluated first, and then a process of substitution of
the modified actual for the formal is invoked.
- The possibility of clash of variables has to be faced. (Cf. the ALCOL 60 report,
sections 4.7.3.2 (Example: \( b \) int \( x \); proc \( P \)(\( x \)); int \( x \); \( b \)...\( e \)...\( P \)(\( x+1 \))...\( e \)) and 4.7.3.3
(Example: \( b \) int \( x \); proc \( P \) ; \( b \)...\( x \)...\( e \)...\( b \) int \( x \)...\( P \)...\( e \)...\( e \)).) These problems are not
exactly the same as encountered in mathematical logic; in particular, they cannot
simply be solved by appropriate use of the notions of free and bound occurrence and
of substitution, as customary in logic.
Section 4 introduces the proof-theoretical framework. It contains the "Exercises in
denotational semantics": For each type of statement, a corresponding axiom or proof
rule is given, and it is required to show its soundness. Also, a modest attempt at
dealing with substitution is included. In fact, for two rules (sequential composition
and conditionals) the proof is easy, for the assignment axiom we refer to [2], whereas
the remaining three cases should, at the moment of writing this, be seen as conjectures
since we do not yet have fully worked out proofs available. However, we are confident
that the rules, perhaps after some minor modifications, will turn out to be sound.

It may be appropriate to add an indication of the restrictions we have imposed
upon our investigation. There are a few minor points (such as: only one procedure de-
claration, i.e., not a simultaneous system; only one parameter of each of the two types,
etc.). Next, things we omitted but which we do not consider essentially difficult (such
as type information in declarations) and, finally, a major omission: We have no func-
tion designators in expressions, nor do we allow procedure identifiers as parameters.

There is a vast amount of literature dealing with the same issues. Many of the
papers take an operational approach, defining semantics in terms of abstract machines.
This we wholly circumvent in the present paper, though it is in fact needed for the justification of the least fixed point approach to recursion (to be given along the lines of De Bakker [1]). Many others take their starting point in some powerful mathematical system (universal algebra, category theory), but tend to fall short of a treatment of the subtler points of the programming notions at hand. A proof-theoretic approach can be found e.g. in Hoare and Wirth [8] or Igarashi, London and Luckham [9], but we must confess not to be able to follow their treatment of procedures and parameter passing. There are also a few papers dealing with the relationship between semantics and proof theory, such as Donahue [4], Cook [3] and Gorelick [5]. Again, the approach of these papers differs from the present one. E.g., the first one omits treatment of recursion, and the other two treat locality in a way which differs from ours (cf. the block rule in our section 4). On the other hand, we recommend the papers by Cook and Gorelick for a discussion of substitution, a topic to which we pay little attention below.

2. SYNTAX

We present a language which is essentially a subset of PASCAL, though there are some notational variants introduced in order to facilitate the presentation. We start with the following classes of symbols:

SV = \{x,y,z,u,\ldots\}: the class of simple variables,
AV = \{a,b,\ldots\} : the class of array variables,
B = \{n,m,\ldots\} : the class of integer constants,
P = \{P,Q,\ldots\} : the class of procedure symbols.

For technical reasons which will become clear below (def. 2.1, def. 3.3), we assume some well-ordering of these four sets.

Using a self-explanatory variant of BNF, we now define the classes V (variables), IE (integer expressions), BE (boolean expressions), and S (statements):

\[
V \quad (v, w, \ldots) : v ::= x \mid a[t]
\]
\[
IE \quad (r, s, t, \ldots) : t ::= v \mid t_1 + t_2 \mid t_1 \times t_2 \mid \text{if } p \text{ then } t_1 \text{ else } t_2 \text{ fi}
\]
\[
BE \quad (p, q, \ldots) : p ::= \text{true} \mid \text{false} \mid t_1 = t_2 \mid t_1 > t_2 \mid p_1 \equiv p_2 \mid p_1 \land p_2 \mid p
\]
\[
S \quad (s, s_0, \ldots) : s ::= v := t \mid s_1 ; s_2 \mid \text{if } p \text{ then } s_1 \text{ else } s_2 \text{ fi}
\]
\[
\begin{align*}
\text{begin new } x \; &; S \end{align*}
\]
\[
\text{end}\; p(t,v).
\]

Remarks

1. We shall use the notation \(t_1 = t_2\) (\(p_1 \equiv p_2\), \(s_1 = s_2\)) to indicate that \(t_1\) and \(t_2\) (\(p_1\) and \(p_2\), \(s_1\) and \(s_2\)) are identical sequences of symbols.
2. Whenever convenient, we shall use parentheses to enhance readability or to avoid ambiguity. Syntactic specification of this is omitted.
3. (Variables) Note that we have simple variables \((x,y,z,u)\) and subscripted variables \((a[t],b[s],\ldots)\), and that an arbitrary variable \(v\) may be both simple or subscripted.
4. (Expressions) The syntax of \( IE \) and \( BE \) has been kept simple on purpose. A minor extension would be to introduce additional operations. On the other hand, the inclusion of functions designators within \( IE \) or \( BE \) presumably would constitute a major extension, requiring substantial additional analysis below.

5. (Statements) In \( S \) we have: assignment, sequential composition, conditionals, blocks, and procedure calls. The last two cases require further comment:

6. (Blocks) We restrict ourselves to declarations of simple variables without type information. This is motivated by our wish to treat declarations only insofar as needed for the analysis of parameter passing.

7. (Procedures) Throughout the paper, we restrict ourselves to the case that we have only one procedure declaration, given in the form

\[
P = \text{val} \; x \cdot \text{var} \; y \cdot S_0
\]

with the following conventions

(a) \( P \in P, \; x, y \in SV, \; S_0 \in S \), with \( x \neq y \).

(b) \( S_0 \) is the procedure body, \( x \) the formal value parameter, \( y \) the formal variable parameter.

(c) In a call \( P(t, v) \), \( t \) is the actual (\( \in IE \)) corresponding to the formal \( x \), and \( v (\in V) \) corresponds to \( y \).

(d) The declaration \( (2.1) \) is assumed to be "globally" available; a call \( P(t, v) \) always refers to \( (2.1) \) as corresponding declaration.

(In PASCAL, one would write for \( (2.1) \):
\[
\text{procedure } P(x: \text{integer}, y: \text{integer}); \text{S}_0 .
\]

Extension to a treatment of systems of declarations is reasonably straightforward (see e.g. \([1]\)), and omitted here mainly for reasons of space; extension to any number of (value and variable) parameters is trivial.

Substitution plays an important role below, both in semantics and proof theory (assignment, locality, parameter mechanisms). In particular, we define

- \( S[v/x] \): substitute the (arbitrary) variable \( v \) for the simple variable \( x \) in \( S \);
- \( s[t/v] \) and \( p[t/v] \): substitute the integer expression \( t \) for the variable \( v \) in \( s \) or \( p \).

The first kind of substitution is defined in the standard way using the notions of free and bound occurrence of a simple variable in a statement (An occurrence of \( x \) in \( S \) is bound whenever it is within a substatement of \( S \) of the form \text{begin new} \; x; \text{S}_1 \; \text{end}. All other occurrences of \( x \) in \( S \) are free.) The second kind of substitution, which includes the case of substitution for a subscripted variable, was introduced in De Bakker \([2]\). We refer to that paper for a detailed account of this, in particular of its application in proving correctness of assignment statements.

DEFINITION 2.1. (Substitution in a statement)

a. \( (w := t)[v/x] \equiv (w[v/x]) := t[v/x] \)

b. \( (S_1; S_2)[v/x] \equiv (S_1[v/x]; S_2[v/x]) \)
c. \((\text{if } p \text{ then } S_1 \text{ else } S_2 \text{ fi})[v/x] = \text{ if } p[v/x] \text{ then } S_1[v/x] \text{ else } S_2[v/x] \text{ fi}\)

d. \((\text{begin new } z; S \text{ end})[v/x] = \text{ begin new } z[v/x] \text{ end, if } x \equiv z\)
\quad \equiv \text{ begin new } z; S[v/x] \text{ end, if } x \not\equiv z \text{ and } z \not\in \text{ free in } v \)
\quad \equiv \text{ begin new } z'; S[z'/z][v/x] \text{ end, if } x \not\equiv z \text{ and } z \in \text{ free in } v, \text{ where } z' \text{ is the first variable } \not\equiv x \text{ not occurring free in } v \text{ or } S\)

e. \(P(t,w)[v/x] = P(t[v/x],w[v/x]).\)

**DEFINITION 2.2. (Substitution in an expression)**

a. The definitions of \(s[t/v] \text{ and } p[t/v]\) are straightforwardly reduced by formula induction to that of \(w[t/v]\), for some \(w \in \mathcal{V} \).

b. We distinguish two cases: \(v \equiv x, \text{ and } v \equiv a[s].\)

   (a) \(x[t/x] = t, \quad y[t/x] = y, \quad a[s][t/x] = a[s[t/x]]\)

   (b) \(x[t/a[s]] = x, \quad b[s'][t/a[s]] = b[s'[t/a[s]]] (a \not\equiv b), \quad a[s'][t/a[s]] = \text{ if } s'[t/a[s]] = s \text{ then } t \text{ else } a[s'[t/a[s]]] \text{ fi}.\)

**Examples**

1. \((\text{begin new } y; \ x:=a[y]; \ P(x+y+z, a[x]) \text{ end})[y/x] = \text{ begin new } y'; \ y:=a[y']; \ P(y+y'+z, a[y]) \text{ end}.\)

2. \(x[1/a[1]] = x, \quad b[2][1/a[1]] = b[2], \quad a[a[2]][1/a[2]] = \text{ if } (\text{if } 2 = a[2] \text{ then } 1 \text{ else } a[2] \text{ fi}) = a[2] \quad \text{ then } 1 \text{ else } a[2] \text{ fi} \text{ fi}.\)

Observe that the last expression is semantically (section 3) (though not syntactically) equal to \(\text{ if } a[2] = 2 \text{ then } a[1] \text{ else } 1 \text{ fi}.\)

3. **DENOTATIONAL SEMANTICS**

   For any two sets \(K, L\), let \((K + L)(K + L)\) denote the set of all functions (all partial functions) from \(K\) to \(L\).

   We define the meaning \(M\) of the various types of statements in our language yielding, for \(S \in \mathcal{S}\), as a result a partial function \(M(S)\) operating on an environment-store pair yielding a new store: \(M(S)(e,o) = o'\).

As starting point we take the set \(A = \{a, a', \ldots\}\) of addresses and the set \(I = \{v, w, \ldots\}\) of integers. Again, we assume these to be well-ordered. Let \(\Sigma = \{\sigma, \sigma', \ldots\}\) be the set of stores, i.e., \(\Sigma = (A + I)\), and let \(\mathbb{Env} = \{\epsilon, \epsilon', \ldots\}\) be the set of environments, i.e., of certain partial, 1-1 functions from \(\mathcal{S} \cup (A \times I)\) to \(A\). More specifically, we require that each \(\epsilon\) is defined on a finite subset of \(\mathcal{S}\), and on all elements \(A \times I\). Thus, for each \(x \in \mathcal{S}\), \(\epsilon(x) \in A\) may be defined, and for each \(a \in A\) and \(v \in I\), \(\epsilon(a,v)\) is defined. (For a subscripted variable \(a[s]\), if \(s\) has the current value \(v\), \(\epsilon(a,v)\) yields the address corresponding to \(a[s]\). The assumption that \(\epsilon(a,v)\) is always defined stems from the fact that we study (explicit) declarations of simple variables only. Array variables may be considered as (implicitly) declared globally.) Next, we
introduce
- For each \( t \in I \) its right-hand value \( R(t)(\varepsilon, \sigma) \in I \),
- For each \( v \in V \) its left-hand value \( L(v)(\varepsilon, \sigma) \in A \),
- For each \( p \in B \) its value \( T(p)(\varepsilon, \sigma) \in \{T, F\} \).

DEFINITION 3.1.
\[ a. \quad R(v)(\varepsilon, \sigma) = \sigma(L(v)(\varepsilon, \sigma)), \quad R(n)(\varepsilon, \sigma) = \nu \text{ (where } \nu \text{ is the integer denoted by the integer constant } n), \]
\[ R(t_1 + t_2)(\varepsilon, \sigma) = plua (R(t_1)(\varepsilon, \sigma), R(t_2)(\varepsilon, \sigma)), \ldots, R(\text{if } p \text{ then } t_1 \text{ else } t_2 \text{ fi})(\varepsilon, \sigma) \]
\[ = \begin{cases} 
R(t_1)(\varepsilon, \sigma), & \text{if } T(p)(\varepsilon, \sigma) = T \\
R(t_2)(\varepsilon, \sigma), & \text{if } T(p)(\varepsilon, \sigma) = F 
\end{cases} \]
\[ b. \quad L(x)(\varepsilon, \sigma) = \sigma(x), \quad L(a[s])(\varepsilon, \sigma) = \sigma(\varepsilon, R(s)(\varepsilon, \sigma)) \]
\[ c. \quad T(\text{true})(\varepsilon, \sigma) = T, \ldots, T(t_1 = t_2)(\varepsilon, \sigma) = \text{equal} (R(t_1)(\varepsilon, \sigma), R(t_2)(\varepsilon, \sigma)), \ldots, \]
\[ T(p_1 \Rightarrow p_2)(\varepsilon, \sigma) = (T(p_1)(\varepsilon, \sigma) \Rightarrow T(p_2)(\varepsilon, \sigma)), \]
where \( \Rightarrow \) denotes implication between the truth-values in \{T, F\}.

For the definition of assignment we need the notion of variant of a store \( \sigma \): We write \( \sigma[v/a] \) for the store which satisfies: \( \sigma[v/a](a) = \nu \), and, for \( a' \neq a \), \( \sigma[v/a](a') = \sigma(a') \).

Using the notations and definitions introduced so far, it is not difficult to define the meaning of the first three types of statements. We shall use the convention that \( M(S)(\varepsilon, \sigma) \) is undefined whenever \( \varepsilon \) is undefined on some variable which occurs free in \( S \) or \( S_0 \). A similar convention applies to \( L, R \) and \( T \).

DEFINITION 3.2. (Assignment, sequential composition, conditionals)
\[ a. \quad M(v:=t)(\varepsilon, \sigma) = \sigma(R(t)(\varepsilon, \sigma)/L(v)(\varepsilon, \sigma)) \]
\[ b. \quad M(S_1; S_2)(\varepsilon, \sigma) = M(S_2)(\varepsilon, M(S_1)(\varepsilon, \sigma)) \]
\[ c. \quad M(\text{if } p \text{ then } S_1 \text{ else } S_2 \text{ fi})(\varepsilon, \sigma) = \begin{cases} 
M(S_1)(\varepsilon, \sigma), & \text{if } T(p)(\varepsilon, \sigma) = T \\
M(S_2)(\varepsilon, \sigma), & \text{if } T(p)(\varepsilon, \sigma) = F. 
\end{cases} \]

For blocks and procedure calls, some further preparations are required. First of all, we require that, for each \( \varepsilon \), \( A \setminus \text{range}(\varepsilon) \) is infinite. Moreover, for each \( \varepsilon \), each \( y \in SV \) not in the domain of \( \varepsilon \), and each \( a \in A \) not in the range of \( \varepsilon \), we use the notation \( \varepsilon \cup \langle y, a \rangle \) for the extension of \( \varepsilon \) defined also on \( y \) (and yielding there \( a \)). This allows us to give

DEFINITION 3.3. (Blocks)
\[ M(\text{begin new } x; S \text{ end})(\varepsilon, \sigma) = M(S[y/x])(\varepsilon \cup \langle y, a \rangle, \sigma), \] where \( y \) is the first variable in \( SV \) not in the domain of \( \varepsilon \), and \( a \) is the first address in \( A \) not in the range of \( \varepsilon \).

The last - and most difficult - case is that of procedure calls. Complications are
- The standard least fixed point treatment of recursion can be given only in terms of a somewhat hybrid entity: a function which expects linguistic objects (elements of \( I \) and \( V \)) as arguments, and yields an element of \( (En \times \varepsilon --\longrightarrow \varepsilon) \) as value.
- The possibility that the actual parameter t has (free) occurrences of the formal x.
- The concept of call-by-variable which, contrary to call-by-name, does not allow straightforward substitution but requires prior evaluation of the subscript in case the actual is a subscripted variable.

Let us consider the declaration $P \equiv \text{val } x \cdot \text{var } y \cdot S_0$, with $S_0 \in S$. In general, $S_0$ will contain "inner" recursive calls of $P$, i.e.,

$$S_0 \equiv \ldots P(t_1,v_1) \ldots \ldots P(t_n,v_n) \ldots$$

Let us use, for any $S \in S$, the notation $S[P+X]$ for the result of replacing all occurrences of $P$ in $S$ by $X$, where $X$ is an element of the set $X = \{X,Y,\ldots\}$ of procedure variables. This result is no longer an element of $S$, but it is easy to extend the definition of $S$ yielding $S_{\text{ext}}$ containing both $S$ and all elements of the form $S[P+X]$.

We shall define the meaning of the procedure $P$ to be an element of a certain subset of the set $H \overset{d}{=} ((IE^xv) \Rightarrow (Env x E \overset{\text{part}}{\Rightarrow} t)))$. In fact, we consider the subset $H_{vi}$ consisting of those elements $\eta$ of $H$ which are variable invariant, i.e., which satisfy $\eta(t[y/x],v[y/x])(\epsilon v<y,a>,s) = \eta(t[y'/x],v[y'/x])(\epsilon v<y',a>,s)$, for all $y,y'$ which do not occur free in $t$, $v$ or $S_0$. Furthermore, we order $H_{vi}$ by putting $\eta \subseteq \eta'$ iff $\forall t,v[v[\eta(t,v)] \leq v'[t,v]]$. Let $\theta,\theta',\ldots$ be elements of the set $(X \Rightarrow H_{vi})$. Thus, it is meaningful to write $\theta(X)(t,v)(\epsilon,s) = \sigma'$. For each $\theta \in (X \Rightarrow H_{vi})$ and each $S_{\text{ext}} \in S_{\text{ext}}'$ we define a mapping $M(\theta)(S_{\text{ext}})$ in the following way:

- For $S_{\text{ext}}$ of one of the first four types, $M(\theta)(S_{\text{ext}})$ is the obvious analogue of $M(S)$.
  E.g., $M(\theta)(v:=t) = \theta(v:=t),\ldots M(\theta)(\text{new } x;S_{\text{ext}})(\epsilon,s) = \theta(\theta(S[y/x]))(\epsilon v<y,a>,s)$, where $y = \ldots$ and $a = \ldots$.
- $M(\theta)(X(t,v)) = \theta(X)(t,v)$.

Actually, we shall mostly use $\theta$'s of the special form $\theta = <X,\eta>$, where we have $<X,\eta>(X) = \eta$, $<X,\eta>(Y)$ is undefined for $X \neq Y$.

Let, for $\phi$ a monotone element of $(H_{vi} \Rightarrow H_{vi})$, $u\phi$ be the least fixed point of $\phi$, i.e., the least element of $H_{vi}$ satisfying $\phi(u\phi) = u\phi$. Let us, finally, write $\text{new } x,y;S_{\text{ext}}$ as short hand for $\text{new } x;\text{new } y;S_{\text{ext}}$, provided that $x \neq y$.

At last, we have enough background to give

**DEFINITION 3.4. (Procedure calls)** Assume the declaration (2.1). Then $M(\theta(t,v)) = (u\phi)(t,v)$, where $\phi$ is the following (monotone) function:

$$\phi = \lambda \eta \cdot \lambda t,v \cdot M(<X,\eta>)$$

$$\begin{align*}
&\text{begin new } u_1, u_2; u_1 := t; u_2 := s; \\
&\text{let } S_0[P:=X][u_1/x][v_1/y] \text{ end}
\end{align*}$$

where $u_1, u_2$ are the first two variables not occurring free in $t$, $v$ or $S_0$, where if $v \equiv z$ for some $z \in SV$, then $s \overset{d}{=} u_2$ and $v_1 \overset{d}{=} z$, where if $v \equiv a[x]$ for some $a \in AV$ and $r \in IE$, then $s \overset{d}{=} r$ and $v_1 \overset{d}{=} a[u_2]$. 

Example. Consider the declaration

\[ P = \nu x. \nu y. \begin{cases} P(x=2, y) \text{ if } x \geq 2 \text{ then } \text{if } x = 1 \text{ then } i:=i+1; P(x-1, a[y]) \text{ else if } x = 1 \text{ then } y:=0 \text{ end if} \end{cases} \]

Then \( M(P(x+5, a[i])) = (\nu i)(x+5, a[i]) \), where we have, e.g.,

\[ \phi(n)(7, y) = M(<X, n>)(b \text{ new } u_1, u_2; u_1 := 7; u_2 := u_2; \text{ if } u_1 \geq 2 \text{ then } P(7, y) \text{ else if } u_1 = 1 \text{ then } i:=i+1; X(u_1-1, a[y] \text{ else } y:=0 \text{ end if}) \]

and

\[ \phi(n)(x+5, a[i]) = M(<X, n>)(b \text{ new } u_1, u_2; u_1 := x+5; u_2 := i; \text{ if } u_1 \geq 2 \text{ then } P(7, a[u_2]) \text{ else if } u_1 = 1 \text{ then } i:=i+1; X(u_1-1, a[u_2]) \text{ else } a[u_2] := 0 \text{ end if}) \]

4. APPLICATIONS TO PROOF THEORY

We introduce the kernel of a system of axioms and proof rules to show the correctness of programs in our PASCAL-like language, and offer as exercises the proofs of the soundness of these axioms and rules.

The formal system is taken from Hoare's axiomatic treatment ([6, 7]) of the inductive assertion method. (Subsequent elaboration of his system may be found e.g. in [8] and [9].)

What we view as our extension of the theory as previously developed, is the following:
- An extension of Hoare's axiom of assignment to the case of assignment to a subscripted variable
- A rule for recursive procedures which extends Scott's induction principle to procedures with call-by-value and call-by-variable parameters.

Let \( p, q \in SE \), \( S \in S_{ext} \). A correctness formula is a construct of the form \( \{p\}S(q) \). Arbitrary correctness formulae are denoted by \( \gamma, \gamma_1, \gamma', \ldots \), and finite sets \( \Gamma = \{\gamma_1, \ldots, \gamma_n\} \) of such formulae are called axioms. Outermost parenthesis in \( \{\gamma_1, \ldots, \gamma_n\} \) are sometimes omitted.

The proof rules of our system are of the following two forms:

\[
\frac{\Gamma_1 \quad \Gamma_2}{\Gamma_3}
\]

DEFINITION 4.1.

a. \( M(\theta)(\Gamma) \) holds iff \( M(\theta)(\gamma) \) holds for each \( \gamma \in \Gamma \).

\( M(\theta)(\{p\}S(q)) \) holds iff for all \( \epsilon \) defined on all free variables of \( p, q, S \) and \( S_0 \), and for all \( \sigma \), we have \( T(p)(\epsilon, \sigma) = T(q)(\epsilon, M(\theta)(S)(\epsilon, \sigma)) \).
b. \( \Gamma \) is valid iff \( M(\Theta)(\Gamma) \) holds for all \( \Theta \).

c. \( \Gamma_1 \vdash \Gamma_2 \) is sound iff, for all \( \Theta \), \( M(\Theta)(\Gamma_1) \) implies \( M(\Theta)(\Gamma_2) \).

d. \( \Gamma_1 \vdash \Gamma_2 \) is sound iff soundness of \( \Gamma_1 \) implies validity of \( \Gamma_3 \).

We now present the axioms and proof rules for the five types of statements and, moreover, a proof rule dealing with substitution. (It is possible to refine the last rule (see [3,5]); however, in the form as given it is sufficiently powerful to allow meaningful application of the procedure rule.)

**Assignment**  
\( \{ p[\{ t/v \}] \} \) \( v := t \{ p \} \).

This axiom, though syntactically identical to Hoare's assignment axiom, is in fact an extension of it since it also covers assignment to subscripted variables.

Example: \( \{ \text{if } a[2] = 2 \text{ then } a[1] = 1 \text{ else true fi} \} \) \( a[2]:=1 \{ a[2]=1 \} \).

**Composition**  
\( \{ p \} S_1 \{ q \} \), \( \{ q \} S_2 \{ r \} \)

\( \{ p \} S_1 \{ r \} \)

**Conditionals**  
\( \{ p \text{ if } q \text{ then } S_1 \text{ else } S_2 \text{ fi} \} \{ r \} \)

These two rules are easily seen to be sound.

**Blocks**  
\( \{ p \} S[y/x]\{ q \} \)

\( \{ p \} \text{ begin new } x; S \text{ end } \{ q \} \)

where \( y \) is some variable which does not occur free in \( p, q, S \) or \( S_0 \).

This rule was first given in Hoare [7]. It is not so easy to grasp all its consequences. Let us point out, e.g., that the fact that it leaves declaration (2.1) unaffected ensures that in a program such as \(< \text{P = var x; val y, ..., z, ..., b new z; ... P(t,v) ... e, e'}} \rangle\), a clash between the global \( z \) of the procedure body, and the local \( z \) valid at the moment of call, is avoided. (As we see it, this problem is incorrectly dealt with in [3,5].)

**Procedure calls.** Assume (2.1), and let \( S_0 \) have the form as described before definition 3.4. Assume we want to prove \( \{ p \} P(t,v) \{ q \} \). Let \( p_0 \overset{df}{=} p \), \( q_0 \overset{df}{=} q \), \( t_0 \overset{df}{=} t \), \( v_0 \overset{df}{=} v \).
\[
\{p_1\} \ X(t_1,v_1) \ {q_1}, \ldots, \{p_n\} \ X(t_n,v_n) \ {q_n}
\]

\rightarrow

\{p_0\} \ \begin{aligned}
\text{begin new } u_1^0, u_2^0, u_3^0 := t_0; u_2^0 := s_0; \\
S_0[P + X][u_1^0/x][v_1^0/y] \ \text{end } \{q_0\}, \\
\end{aligned}

\ldots

\{p_n\} \ \begin{aligned}
\text{begin new } u_1^n, u_2^n, u_3^n := t_n; u_2^n := s_n; \\
S_0[P + X][u_1^n/x][v_1^n/y] \ \text{end } \{q_n\}
\end{aligned}

\{p_0\} \ P(t_0,v_0) \ {q_0}

where, for each \( i = 0,1,\ldots,n \), the \( u_1^i, u_2^i \) do not occur free in \( S_0, t_i, v_i, p_i \) or \( q_i \), and where the \( s_i \) and \( v_i \), \( i = 0,\ldots,n \), are derived from the \( v_i \) in the same manner as in def. 3.4.

Observe that the \( p_i, q_i \), \( i = 1,\ldots,n \), are assertions about the inner calls, whereas the \( p_0, q_0 \) are assertions about the outer call. Therefore, the \( p_0, q_0 \) do not play a part in the induction hypothesis. One should also observe that the rule remains valid when the formulae \( \{p_i\} \ P(t_i,v_i) \ {q_i}, \ i = 1,\ldots,n \), are added to its conclusion (i.e. to \( \{p_0\} \ P(t_0,v_0) \ {q_0} \)).

**Substitution**

\[
\begin{aligned}
P & \leftarrow \text{var } x \cdot \text{val } y \cdot S_0, \ (p)S(q) \\
& \leftarrow (\text{var } x \cdot \text{val } y \cdot S_0)[v/u], (p[v/u])S[v/u](q[v/u])
\end{aligned}
\]

where \( v \) satisfies the following requirement: None of the simple variables occurring in \( v \) occur free in \( S, S_0, p \) or \( q \).

We hope that the notation in this rule - which extends the definitions given so far - is self-explanatory: Above the line, calls of \( P \) refer to declaration (2.1), but below they refer to the declaration \( P \leftarrow (\text{var } x \cdot \text{val } y \cdot S_0)[v/u] \), where a natural extension of def. 2.1 is assumed.

We are confident that the proofs of the soundness of the block rule, the procedure call rule and the substitution rule, will offer no difficulties.
REFERENCES


