EXERCISES IN DENOTATIONAL SEMANTICS

K.R. Apt

J.W. de Bakker

Mathematisch Centrum, Amsterdam

1. 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]), in sofar 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 language.

The paper has the following outline:

Section 2 gives the syntax of the various language constructs. Also, a careful definition of *substitution* is given which is needed for the treatment of assignment, locality 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* ε (a mapping from variables to addresses) and *store* σ (a mapping from addresses to values), yielding a new store σ' : $M(S)(\varepsilon,\sigma) = \sigma'$. For assignment, sequential composition and conditionals the definitions are fairly straightforward. It is also reasonably clear what to do about locality, but the treatment of procedures 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 ALGOL 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 ovt 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 declaration, 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 function 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:

$\mathcal{SV} = \{x,y,z,u,\dots\}$: the class of *simple variables*,
$\mathcal{AV} = \{a,b,\dots\}$: the class of *array variables*,
$\mathcal{B} = \{n,m,\dots\}$: the class of *integer constants*,
$\mathcal{P} = \{P,Q,\dots\}$: 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 $\mathcal{V}$ (*variables*), $\mathcal{IE}$ (*integer expressions*), $\mathcal{BE}$ (*boolean expressions*), and $\mathcal{S}$ (*statements*):

$\mathcal{V}$ (with elements $v,w,\dots$) $v ::= x \mid a[t]$
$\mathcal{IE}$ (with elements $r,s,t,\dots$) $t ::= v \mid n \mid t_1+t_2 \mid t_1*t_2 \mid \underline{\text{if}}\ p\ \underline{\text{then}}\ t_1\ \underline{\text{else}}\ t_2\ \underline{\text{fi}}$
$\mathcal{BE}$ (with elements $p,q,\dots$) $p ::= \underline{\text{true}} \mid \underline{\text{false}} \mid t_1=t_2 \mid t_1>t_2 \mid p_1 \supset p_2 \mid p_1 \wedge p_2 \mid \neg p$
$\mathcal{S}$ (with elements $S,S_0,\dots$) $S ::= v:=t \mid S_1;S_2 \mid \underline{\text{if}}\ p\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}} \mid$
$\underline{\text{begin}}\ \underline{\text{new}}\ x;\ S\ \underline{\text{end}} \mid P(t,v)$.

Remarks

1. We shall use the notation $t_1 \equiv t_2$ ($p_1 \equiv p_2$, $S_1 \equiv S_2$) 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 ($a[t],b[s],\dots$), 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 in sofar 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

(2.1) $\quad P \Leftarrow \mathit{val}\ x \cdot \mathit{var}\ y \cdot S_0$

with the following conventions

(α) $P \in P$, $x,y \in SV$, $S_0 \in S$, with $x \not\equiv y$.

(β) S_0 is the *procedure body*, x the formal value parameter, y the formal variable parameter.

(γ) In a *call* P(t,v), t is the actual ($\in \mathit{IE}$) corresponding to the formal x, and v ($\in V$) corresponds to y.

(δ) 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):
<u>procedure</u> P(x:integer,<u>var</u> y:integer);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 <u>begin</u> <u>new</u> x;S_1 <u>end</u>. 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. (if p then S_1 else S_2 fi)[v/x] ≡ if p[v/x] then S_1[v/x] else S_2[v/x] fi

d. (begin new z;S end)[v/x] ≡ begin new z;S end, if x ≡ z
≡ begin new z;S[v/x] end, if x ≢ z and z does not occur free in v
≡ begin new z';S[z'/z][v/x] end, if x ≢ z and z occurs free in v, where z' is the first variable ≢ x not occurring free in v 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] and p[t/v] are straightforwardly reduced by formula induction to that of w[t/v], for some w ∈ V.

b. We distinguish two cases: v ≡ x, and v ≡ a[s].
(α) x[t/x] ≡ t, y[t/x] ≡ y (x≢y), a[s][t/x] ≡ a[s[t/x]]
(β) x[t/a[s]]≡ x, b[s'][t/a[s]] ≡ b[s'[t/a[s]]] (a≢b),
a[s'][t/a[s]] ≡ if s'[t/a[s]] = s then t else a[s'[t/a[s]]] fi.

Examples

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

2. x[1/a[a[1]]] ≡ x, b[2][1/a[a[1]]] ≡ b[2],
a[a[2]][1/a[a[2]]] ≡ if(if 2 = a[2] then 1 else a[2] fi) = a[2]
then 1 else a[if 2 = a[2] then 1 else a[2] fi] fi.
Observe that the last expression is semantically (section 3) (though not syntactically) equal to if a[2] = 2 then a[1] else 1 fi.

3. DENOTATIONAL SEMANTICS

For any two sets K, L, let $(K \to L)$ $((K \xrightarrow[\text{part}]{} L))$ 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 ∈ S, as a result a partial function M(S) operating on an environment-store pair yielding a new store: $M(S)(\varepsilon,\sigma) = \sigma'$.

As starting point we take the set $A = \{\alpha,\beta,\ldots\}$ of *addresses* and the set $I = \{\nu,\mu,\ldots\}$ of *integers*. Again, we assume these to be well-ordered. Let $\Sigma = \{\sigma,\sigma',\ldots\}$ be the set of *stores*, i.e. $\Sigma = (A \to I)$, and let $Env = \{\varepsilon,\varepsilon',\ldots\}$ be the set of *environments*, i.e., of certain *partial*, 1 - 1 functions from $SV \cup (AV \times I)$ to A. More specifically, we require that each ε is defined on a *finite* subset of SV, and on *all* elements $AV \times I$. Thus, for each x ∈ SV, ε(x) ∈ A may be defined, and for each a ∈ AV and ν ∈ I, ε(a,ν) *is* defined. (For a subscripted variable a[s], if s has the current value ν, ε(a,ν) yields the address corresponding to a[s]. The assumption that ε(a,ν) 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 $t \in \mathit{IE}$ 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 $p \in \mathit{BE}$ its *value* $T(p)(\varepsilon,\sigma) \in \{T,F\}$.

DEFINITION 3.1.

a. $R(v)(\varepsilon,\sigma) = \sigma(L(v)(\varepsilon,\sigma))$, $R(n)(\varepsilon,\sigma) = \nu$ (where ν is the integer denoted by the integer constant n), $R(t_1+t_2)(\varepsilon,\sigma) = \mathit{plus}\ (R(t_1)(\varepsilon,\sigma),R(t_2)(\varepsilon,\sigma)),\ldots,R(\underline{\text{if}}\ p\ \underline{\text{then}}\ t_1\ \underline{\text{else}}\ t_2\ \underline{\text{fi}})(\varepsilon,\sigma)$

$$= \begin{cases} R(t_1)(\varepsilon,\sigma), & \text{if } T(p)(\varepsilon,\sigma) = T \\ R(t_2)(\varepsilon,\sigma), & \text{if } T(p)(\varepsilon,\sigma) = F \end{cases}$$

b. $L(x)(\varepsilon,\sigma) = \varepsilon(x)$, $L(a[s])(\varepsilon,\sigma) = \varepsilon(a,R(s)(\varepsilon,\sigma))$

c. $T(\underline{\text{true}})(\varepsilon,\sigma) = T,\ldots,T(t_1=t_2)(\varepsilon,\sigma) = \mathit{equal}\ (R(t_1)(\varepsilon,\sigma),R(t_2)(\varepsilon,\sigma)),\ldots,$
$T(p_1 \supset p_2)(\varepsilon,\sigma) = (T(p_1)(\varepsilon,\sigma) \Rightarrow T(p_2)(\varepsilon,\sigma))$,
where "$\Rightarrow$" denotes implication between the truth-values in $\{T,F\},\ldots$.

For the definition of assignment we need the notion of *variant* of a store σ: We write $\sigma\{\nu/\alpha\}$ for the store which satisfies: $\sigma\{\nu/\alpha\}(\alpha) = \nu$, and, for $\alpha' \neq \alpha$, $\sigma\{\nu/\alpha\}(\alpha') = \sigma(\alpha')$.

Using the notations and definitions introduced sofar, 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 ε 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. $M(v{:=}t)(\varepsilon,\alpha) = \sigma\{R(t)(\varepsilon,\sigma)/L(v)(\varepsilon,\sigma)\}$

b. $M(S_1;S_2)(\varepsilon,\sigma) = M(S_2)(\varepsilon,M(S_1)(\varepsilon,\sigma))$

c. $M(\underline{\text{if}}\ p\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}) = \begin{cases} M(S_1)(\varepsilon,\sigma), & \text{if } T(p)(\varepsilon,\sigma) = T \\ M(S_2)(\varepsilon,\sigma), & \text{if } T(p)(\varepsilon,\sigma) = F. \end{cases}$

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

DEFINITION 3.3. (Blocks)

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

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 IE and V) as arguments, and yields an element of $(\mathit{Env} \times \Sigma \xrightarrow[\text{part}]{} \Sigma)$ 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 $P \Leftarrow \mathit{val}\ x \cdot \mathit{var}\ y \cdot S_0$, with $S_0 \in \mathcal{S}$. In general, S_0 will contain "inner" recursive calls of P, i.e.,

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

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

We shall define the meaning of the procedure P to be an element of a certain subset of the set $\mathcal{H} \overset{\text{df.}}{\equiv} ((\mathit{IE}\times\mathcal{V}) \to (\mathit{Env}\times\Sigma \xrightarrow[\text{part}]{} \Sigma))$. In fact, we consider the subset $\mathcal{H}_{vi}$ consisting of those elements η of $\mathcal{H}$ which are *variable invariant*, i.e., which satisfy $\eta(t[y/x],v[y/x])(\varepsilon\cup\langle y,\alpha\rangle,\sigma) = \eta(t[y'/x],v[y'/x])(\varepsilon\cup\langle y',\alpha\rangle,\sigma)$, for all y,y' which do not occur free in t, v or S_0. Furthermore, we order $\mathcal{H}_{vi}$ by putting $\eta \sqsubseteq \eta'$ iff $\forall t,v[\eta(t,v) \sqsubseteq \eta'(t,v)]$. Let $\theta,\theta',\ldots$ be elements of the set $(\mathcal{X}\to\mathcal{H}_{vi})$. Thus, it is meaningful to write $\theta(X)(t,v)(\varepsilon,\sigma) = \sigma'$. For each $\theta \in (\mathcal{X}\to\mathcal{H}_{vi})$ and each $S_{ext} \in \mathcal{S}_{ext}$, we define a mapping $\mathcal{M}(\theta)(S_{ext})$ in the following way:

- For S_{ext} of one of the first four types, $\mathcal{M}(\theta)(S_{ext})$ is the obvious analogue of $\mathcal{M}(S)$. E.g., $\mathcal{M}(\theta)(v:=t) = \mathcal{M}(v:=t),\ldots,\mathcal{M}(\theta)(\underline{b}\ \underline{\text{new}}\ x;S\ \underline{e})(\varepsilon,\sigma) = \mathcal{M}(\theta)(S[y/x])(\varepsilon\cup\langle y,\alpha\rangle,\sigma)$, where $y = \ldots$ and $\alpha = \ldots$.
- $\mathcal{M}(\theta)(X(t,v)) = \theta(X)(t,v)$.

Actually, we shall mostly use θ's of the special form $\theta = \langle X,\eta\rangle$, where we have $\langle X,\eta\rangle(X) = \eta$, $\langle X,\eta\rangle(Y)$ is undefined for $X \not\equiv Y$.

Let, for Φ a monotone element of $(\mathcal{H}_{vi} \to \mathcal{H}_{vi})$, $\mu\Phi$ be the least fixed point of Φ, i.e., the least element of $\mathcal{H}_{vi}$ satisfying $\Phi(\mu\Phi) = \mu\Phi$. Let us, finally, write $\underline{b}\ \underline{\text{new}}\ x,y;S\ \underline{e}$ as short hand for $\underline{b}\ \underline{\text{new}}\ x;\underline{b}\ \underline{\text{new}}\ y;S\ \underline{e}\ \underline{e}$, provided that $x \not\equiv y$.

At last, we have enough background to give

DEFINITION 3.4. (Procedure calls) Assume the declaration (2.1). Then $\mathcal{M}(P(t,v)) = (\mu\Phi)(t,v)$, where Φ is the following (monotone) function:

$$\begin{aligned}\Phi = \lambda\eta\cdot\lambda t,v\cdot\ &\mathcal{M}(\langle X,\eta\rangle)\\ &(\underline{\text{begin}}\ \underline{\text{new}}\ u_1,u_2;u_1:=t;u_2:=s;\\ &\quad S_0[P\to X][u_1/x][v_1/y]\ \underline{\text{end}})\end{aligned}$$

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 $z \in \mathcal{SV}$, then $s \overset{\text{df.}}{\equiv} u_2$ and $v_1 \overset{\text{df.}}{\equiv} z$,
where if $v \equiv a[r]$ for some $a \in \mathcal{AV}$ and $r \in \mathit{IE}$, then $s \overset{\text{df.}}{\equiv} r$ and $v_1 \overset{\text{df.}}{\equiv} a[u_2]$.

Example. Consider the declaration

$$P \Leftarrow \mathit{val}\ x \cdot \mathit{var}\ y \cdot \underline{\mathrm{if}}\ x \geq 2\ \underline{\mathrm{then}}\ P(7,y)\ \underline{\mathrm{else}}\ \underline{\mathrm{if}}\ x = 1\ \underline{\mathrm{then}}$$
$$i{:}{+}i{+}1; P(x-1,a[y])\ \underline{\mathrm{else}}\ y{:=}0\ \underline{\mathrm{fi}}\ \underline{\mathrm{fi}}\ .$$

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

$$\Phi(\eta)(7,y) = M(<X,\eta>)(\underline{\mathrm{b}}\ \underline{\mathrm{new}}\ u_1,u_2;u_1{:=}7;u_2{:=}u_2;\ \underline{\mathrm{if}}\ u_1 \geq 2\ \underline{\mathrm{then}}\ P(7,y)\ \underline{\mathrm{else}}\ \underline{\mathrm{if}}\ u_1 = 1\ \underline{\mathrm{then}}$$
$$i{:=}i{+}1; X(u_1-1,a[y]\ \underline{\mathrm{else}}\ y{:=}0\ \underline{\mathrm{fi}}\ \underline{\mathrm{fi}}\ \underline{\mathrm{e}})$$

and

$$\Phi(\eta)(x+5,a[i]) = M(<X,\eta>)(\underline{\mathrm{b}}\ \underline{\mathrm{new}}\ u_1,u_2;u_1{:=}x+5;u_2{:=}i;\ \underline{\mathrm{if}}\, u_1 \geq 2\ \underline{\mathrm{then}}\, P(7,a[u_2]\ \underline{\mathrm{else}}\ \underline{\mathrm{if}}\, u_1 = 1$$
$$\underline{\mathrm{then}}$$
$$i{:=}i{+}1; X(u_1-1,a[a[u_2]])\ \underline{\mathrm{else}}\ a[u_2]{:=}0\ \underline{\mathrm{fi}}\ \underline{\mathrm{fi}}\ \underline{\mathrm{e}}).$$

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 $p,q \in BE$, $S \in S_{ext}$. 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:

$$(4.1) \qquad \frac{\Gamma_1}{\Gamma_2}$$

$$(4.2) \qquad \frac{\Gamma_1 \rightarrow \Gamma_2}{\Gamma_3}$$

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 ε defined on all free variables of p,q,S and S_0, and for all σ, we have $T(p)(\varepsilon,\sigma) \Rightarrow T(q)(\varepsilon,M(\theta)(S)(\varepsilon,\sigma))$.

b. Γ is valid iff $M(\theta)(\Gamma)$ holds for all θ.

c. $\frac{\Gamma_1}{\Gamma_2}$ is sound iff, for all θ, $M(\theta)(\Gamma_1)$ implies $M(\theta)(\Gamma_2)$.

d. $\frac{\Gamma_1 \rightarrow \Gamma_2}{\Gamma_3}$ is sound iff soundness of $\frac{\Gamma_1}{\Gamma_2}$ implies validity of Γ_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: $\{\underline{\text{if}}\ a[2] = 2\ \underline{\text{then}}\ a[1] = 1\ \underline{\text{else}}\ \underline{\text{true}}\ \underline{\text{fi}}\}\ a[a[2]]{:=}1\ \{a[a[2]]=1\}$. For details see [2].

Composition
$$\frac{\{p\}S_1\{q\},\{q\}S_2\{r\}}{\{p\}S_1;S_2\{r\}}$$

Conditionals
$$\frac{\{p\wedge q\}S_1\{r\},\{p\wedge\neg q\}S_2\{r\}}{\{p\}\ \underline{\text{if}}\ q\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}\ \{r\}}$$

These two rules are easily seen to be sound.

Blocks
$$\frac{\{p\}\ S[y/x]\ \{q\}}{\{p\}\ \underline{\text{begin}}\ \underline{\text{new}}\ x;S\ \underline{\text{end}}\ \{q\}}$$

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 $< P \Leftarrow$ *var* x $\cdot$ *val* y $\cdot$ ... z ..., $\underline{\text{b}}$ ... $\underline{\text{b}}$ $\underline{\text{new}}$ z; ... P(t,v) ... $\underline{\text{e}}$ ... $\underline{\text{e}} >$, 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 $p_0 \overset{\text{df.}}{\equiv} p$, $q_0 \overset{\text{df.}}{\equiv} q$, $t_0 \overset{\text{df.}}{\equiv} t$, $v_0 \overset{\text{df.}}{\equiv} v$.

$$\{p_1\}\ X(t_1,v_1)\ \{q_1\},\ldots,\{p_n\}\ X(t_n,v_n)\ \{q_n\}$$

$$\rightarrow$$

$$\{p_0\}\ \underline{\text{begin}}\ \underline{\text{new}}\ u_1^0,u_2^0,u_1^0{:=}t_0;u_2^0{:=}s_0;$$

$$S_0[P\rightarrow X][u_1^0/x][v_1^0/y]\ \underline{\text{end}}\ \{q_0\},$$

$$\ldots$$

$$\{p_n\}\ \underline{\text{begin}}\ \underline{\text{new}}\ u_1^n,u_2^n;u_1^n{:=}t_n;u_2^n{:=}s_n;$$

$$S_0[P\rightarrow X][u_1^n/x][v_1^n/y]\ \underline{\text{end}}\ \{q_n\}$$

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

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_1^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

$$\frac{P \Leftarrow \mathit{var}\ x \cdot \mathit{val}\ y \cdot S_0,\ \{p\}S\{q\}}{P \Leftarrow (\mathit{var}\ x \cdot \mathit{val}\ y \cdot S_0)[v/u],\{p[v/u]\}S[v/u]\{q[v/u]\}}$$

where v *satisfies the following requirement: None of the simple variables occurring in* v *occurs free in* S, S_0, p *or* q.

We hope that the notation in this rule - which extends the definitions given sofar - is self-explanatory: Above the line, calls of P refer to declaration (2.1), but below they refer to the declaration $P \Leftarrow (\mathit{var}\ x \cdot \mathit{val}\ y \cdot S_0)[v/u]$, 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

1. De Bakker, J.W., *Least fixed points revisited,* in λ-Calculus and Computer Science Theory, Lecture Notes in Computer Science 37 (C. Böhm, ed.), p.27-61, Springer (1975).

2. De Bakker, J.W., *Correctness proofs for assignment statements*, Report IW 55/76, Mathematisch Centrum (1976).

3. Cook, S.A., *Axiomatic and interpretive semantics for an ALGOL fragment*, Technical Report no. 79, University of Toronto (1975).

4. Donahue, J.E., *The mathematical semantics of axiomatically defined programming language constructs*, in Proc. Symp. Proving and Improving Programs, p.353-370, IRIA (1975).

5. Gorelick, G.A., *A complete axiomatic system for proving assertions about recursive and non-recursive programs*, Technical Report no. 75, University of Toronto (1975).

6. Hoare, C.A.R., *An axiomatic basis for programming language constructs*, C.ACM 12, p.576-580 (1969).

7. Hoare, C.A.R., *Procedures and parameters, an axiomatic approach,* in Symp. on Semantics of Algorithmic Languages, Lecture Notes in Mathematics 188 (E. Engeler, ed.), p.102-116, Springer (1971).

8. Hoare, C.A.R. & N. Wirth, *An axiomatic definition of the programming language PASCAL*, Acta Inf. 2, p.335-355 (1973).

9. Igarashi, S., R.L. London & D.C. Luckham, *Automatic program verification I: A logical basis and its implementation*, Acta Inf. 4, p.145-182 (1975).

10. Manna, Z., S. Ness & J. Vuillemin, *Inductive methods for proving properties of programs*, C.ACM 16, p.491-502 (1973).

11. Manna, Z. & J. Vuillemin, *Fixpoint approach to the theory of computation*, C.ACM 15, p.528-536 (1972).

12. Scott, D. & C. Strachey, *Towards a mathematical semantics for computer languages*, in Proc. of the Symp. on Computers and Automata (J. Fox, ed.), p.19-46, Polytechnic Inst. of Brooklyn (1971).