Reprinted from JWb2

INFORMATION PROCESSING 77

Proceedings of IFIP Congress 77
Toronto, August 8-12, 1977

Edited by

Bruce GILCHRIST
Columbia University
New York

PROGRAM COMMITTEE

1977

NORTH-HOLLAND PUBLISHING COMPANY
AMSTERDAM · NEW YORK · OXFORD

INFORMATION PROCESSING 77, B. GILCHRIST, EDITOR

SEMANTICS AND THE FOUNDATIONS OF PROGRAM PROVING

J. W. DE BAKKER
Mathematical Centre & Free University
Amsterdam, The Netherlands

(INVITED PAPER)

A discussion is presented of some of the applications of mathematical (also called denotational) semantics in the justification of a proof theory for program correctness. Syntax and (denotational) semantics of a simple example language are given, together with a sketch of the associated proof theory which is rather economic in the structure of its assertions. The system is applied to three case studies in program proving: assignment to a subscripted variable, weakest preconditions and the while statement, and the parameter mechanisms of PASCAL. An appendix contains further details on the while statement.

1. INTRODUCTION

We see as a major task for theoretical computer science the development of a mathematical theory of programming languages, aimed at a better understanding of the fundamental notions in programming, and, hopefully, resulting in an improved quality of their applications. In our paper we will present a review of some of the current issues in this area, with the main emphasis on the interface between semantics and program correctness proofs.

Let us first briefly indicate in which sense we want to take these terms. As usual in language theory, we distinguish between problems of *form* and *content*, the former corresponding to the study of *syntax* - how to specify and analyse well-formed programs - the latter leading us into the realm of *semantics*, where we study ways of attributing *meaning* to programs.

Unfortunately, there is no agreement at all on what constitutes a proper methodology for semantic specification. On the contrary, we find ourselves confronted with an embarrassingly rich choice of approaches, ranging from the simple view that a language is best defined through its compiler, via intriguing applications of various form of modal logic, to the use of sophisticated techniques rooted in category theory or universal algebra.

We find it advantageous to distinguish three main trends in the field of semantic description of programming languages. Two of these are what one might call model-theoretic, in the sense that meaning is attributed to programs by relating them to a model, i.e., some universe which is not the same as the linguistic world of the program texts. Of course, the same idea applies to natural languages: A linguistic object - for example, the word "table" which happens to consist of five letters - is assigned meaning through its correspondence to the external world - where we might observe a table as an object with four legs. For many years, the only universe used in the specification of the meaning of programs was that of a - real or abstract - *machine*. In this point of view, each program instruction determines a *state-transforming* action of the machine, and execution of a complete program leads to a sequence of states, starting from an initial state and, normally, terminating in some final state. It has become customary to refer to this as *operational semantics*. Important examples of it are the definition of PL/I with the so-called Vienna method, [19] and the definition of ALGOL 68. [32] In recent years, a second model-theoretic approach has gained increasing acceptance, namely the method of *mathematical* (or *denotational*) *semantics* advocated by the Oxford school of Dana Scott and the late Christopher Strachey [29] (see also, e.g., [21,31]). The qualification "mathematical" is here not to be taken as implying that the methods of operational semantics would not necessarily satisfy mathematical standards. Rather, it reflects the nature of the model used, which is completely machine-independent and relies solely on certain basic mathematical notions such as sets, functions and operators. Since we will make extensive use of these ideas in the technical development below, we will not go into details now. The third group of techniques used in the study of languages is *proof-theoretic* - as opposed to the model-theoretic nature of the first two. As an implicit way of assigning meaning to programs, one proposes certain axioms and proof rules which are used in the (formal) proofs of program properties. As an outstanding representative of this approach we mention the inductive assertion method, originally proposed by Floyd, [13] embedded in a formal system by Hoare, [14] and reappearing in somewhat modified form in Dijkstra's work on weakest preconditions. [12]

In our opinion, care should be taken not to view these three methodologies as competetive ones, but, on the contrary, as complementary in that no single one of them is appropriate for all possible applications. The remainder of our paper will be devoted to an illustration of how mathematical semantics can help in clarifying proof theory. However, let us emphasize that operational semantics has just as an important role in that it is closest to the actual problems of the compiler writer.

Let us now outline how the rest of the paper is organized. We first present a very simple language and define its mathematical semantics. Next, we state the sort of formal assertions one might be interested to make on this language, and sketch the structure of a possible proof theory for it. We then proceed with three applications dealing with

- assignment, in particular to subscripted variables
- weakest preconditions and the while statement
- the parameter mechanisms call-by-value and call-by-variable, as occurring in the language PASCAL.

We hope to show what challenges are offered to mathematical semantics by this sample of problems in the area of program proving. Though the examples treated are simple, we find that they are not always well-understood. It has been our experience that the foundations of program proving are in danger of being somewhat shaky, when established without the support

of semantic justification.
(Related investigations of the connections between semantics and proof theory have been reported for example by Donahue, [11] Ligler, [17,18] and Pratt. [26] (Cf. also Milner [22].)

2. SYNTAX AND SEMANTICS OF A SIMPLE LANGUAGE

Our example language has three kinds of constructs, viz. *statements*, *integer expressions*, and *boolean expressions*. As the starting point in the formation of integer expressions we take the classes of *integer variables* $Var = \{x,y,\ldots\}$ and of *integer constants* $Const = \{m,n,\ldots\}$. Using a syntactic definition formalism which should be self-explanatory, we then introduce:

The class of statements *Stat* with elements S,...

$S ::= x{:=}s \mid S_1;S_2 \mid \underline{if}\ b\ \underline{then}\ S_1\ \underline{else}\ S_2\ \underline{fi} \mid \underline{while}\ b\ \underline{do}\ S\ \underline{od}$

The class of integer expressions *Iexp* with elements s,t,...

$s ::= x \mid m \mid s_1{+}s_2 \mid \underline{if}\ b\ \underline{then}\ s_1\ \underline{else}\ s_2\ \underline{fi}$

The class of boolean expressions *Bexp* with elements b,...

$b ::= \underline{true} \mid \underline{false} \mid s_1{=}s_2 \mid \neg b \mid b_1 \supset b_2$

Meaning is attributed to the constructs of this language with respect to a *state*, i.e., a mapping from variables to values. E.g., the meaning of the assignment statement x:= x+1 in a state where x has the value 0 is a new state in which x now has the value 1 (and all other variables have maintained their old values).

Let $I = \{\mu,\nu,\ldots\}$ be the set of integers (note that in our programming language we use integer constants in *Const* to denote these), and let $\Sigma = Var \to I$ be the set of states, with elements $\sigma,\sigma',\ldots$. We now introduce mappings M, V and T, defining the meaning of the elements in *Stat*, *Iexp* and *Bexp*, respectively, all with respect to a given state:

$$M\colon Stat \to (\Sigma \xrightarrow[\text{part}]{} \Sigma)$$

$$V\colon Iexp \to (\Sigma \longrightarrow I)$$

$$T\colon Bexp \to (\Sigma \longrightarrow \{T,F\}).$$

These definitions should be read as follows: For each statement S, $M(S)$ yields a (partial) function from states to states (thus, it is meaningful to write $M(S)(\sigma) = \sigma'$). Similarly, for each s, $V(s)$ yields a function from states to integers (we can write $V(s)(\sigma) = \mu$), and $T(b)$ yields a function from states to the set consisting of the two truth-values T and F (e.g., $T(b)(\sigma) = T$ might hold).

Before presenting the semantic definitions, we present one further piece of notation: For $\sigma \in \Sigma$, $x \in Var$ and $\mu \in I$, we define $\sigma\{\mu/x\}$ as a new state given by: $\sigma\{\mu/x\}(x) = \mu$, and for each $y \not\equiv x$: $\sigma\{\mu/x\}(y) = \sigma(y)$.

This formalism enables us to give a succinct definition of the concepts in our simple language. For each σ:

$$M(x{:=}s)(\sigma) = \sigma\{V(s)(\sigma)/x\}$$

$$M(S_1;S_2)(\sigma) = M(S_2)(M(S_1)(\sigma))$$

$$M(\underline{if}\ b\ \underline{then}\ S_1\ \underline{else}\ S_2\ \underline{fi})(\sigma) = \begin{cases} M(S_1)(\sigma) \text{ if } T(b)(\sigma) = T \\ M(S_2)(\sigma) \text{ if } T(b)(\sigma) = F \end{cases}$$

$M(\underline{while}\ b\ \underline{do}\ S\ \underline{od})(\sigma) =$ (this case is somewhat more complex than the other ones, and relegated to the Appendix)

$$V(x)(\sigma) = \sigma(x)$$

$V(m)(\sigma) = \mu$ (the integer denoted by the constant m)

$V(s_1{+}s_2)(\sigma) = plus\ (V(s_1)(\sigma), V(s_2)(\sigma))$ (where we assume known the meaning of the mathematical function $plus\colon I \times I \to I$)

$$V(\underline{if}\ b\ \underline{then}\ s_1\ \underline{else}\ s_2\ \underline{fi})(\sigma) = \begin{cases} V(s_1)(\sigma) \text{ if } T(b)(\sigma) = T \\ V(s_2)(\sigma) \text{ if } T(b)(\sigma) = F \end{cases}$$

$$T(\underline{true})(\sigma) = T$$

$$T(\underline{false})(\sigma) = F$$

$T(s_1{=}s_2)(\sigma) = equal\ (V(s_1)(\sigma), V(s_2)(\sigma))$ (where we assume known the meaning of the mathematical function $equal\colon I \times I \to \{T,F\}$)

$$T(\neg b)(\sigma) = \begin{cases} F, \text{ if } T(b)(\sigma) = T \\ T, \text{ if } T(b)(\sigma) = F \end{cases}$$

$T(b_1 \supset b_2)(\sigma) = (T(b_1)(\sigma) \Rightarrow T(b_2)(\sigma))$ (where we assume known the meaning of the logical operation "$\Rightarrow$" between truth-values).

Examples. First we determine $M(x{:=}x)(\sigma)$ as follows: $M(x{:=}x)(\sigma) = \sigma\{V(x)(\sigma)/x\} = \sigma\{\sigma(x)/x\} = \sigma$. (Below, we shall use Δ as the abbreviation for the "dummy statement" x:=x.) Next, we evaluate $M(x{:=}2;y{:=}x{+}y)(\sigma)$, where σ satisfies $\sigma(y) = 1$. We obtain successively - neglecting for the moment the distinction between integer constants and integers:

$$M(x{:=}2;y{:=}x{+}y)(\sigma) =$$
$$M(y{:=}x{+}y)(M(x{:=}2)(\sigma)) =$$
$$M(y{:=}x{+}y)(\sigma\{2/x\}) =$$
$$\sigma\{2/x\}\{plus\ (V(x)(\sigma\{2/x\}), V(y)(\sigma\{2/x\}))/y\} =$$
$$\sigma\{2/x\}\{plus(2,1)/y\} =$$
$$\sigma\{2/x\}\{3/y\}.$$

Once having acquired some familiarity with the notation, the reader will easily convince himself that the definitions indeed capture the usual meaning of the concepts in our language. Of course, the definitions become considerably more complex for more interesting languages, but, still, the basic approach remains essentially the same as the one described here.

3. PROOF THEORY

Proofs about programs are usually concerned with three types of program properties:

- *correctness*: Program S is correct if and only if it transforms input satisfying condition p_1 to output satisfying condition p_2, for suitably chosen conditions p_1,p_2.
- *termination*: The computation specified by program S terminates for all input satisfying a suitable condition p.
- *equivalence*: Programs S_1 and S_2 determine the same state transformation.

We shall outline a formal system in which these properties can be formulated for our simple language, together with a definition of the notion of justifying the system using the semantics as given in section 2.

The *formulae* of the system are either *assertions* or *equivalences*. The class of assertions p,q,... is an extension of the class of boolean expressions *Bexp* of section 2:

$$p ::= \underline{\text{true}} \mid \underline{\text{false}} \mid s_1 = s_2 \mid \neg p \mid p_1 \supset p_2 \mid S;p \mid \exists x[p]$$

An equivalence is a construct of the form $S_1 = S_2$. We now extend the function T to assertions and equivalences. Thus, its definition for the first five syntactic clauses in the syntax for p is just as before, and therefore is not repeated. Furthermore, we define, for each σ,

$$T(S;p)(\sigma) = \begin{cases} T, & \text{if there exists } \sigma' \text{ such that } \sigma' = M(S)(\sigma) \text{ and } T(p)(\sigma') = T \\ F, & \text{otherwise.} \end{cases}$$

$$T(\exists x[p])(\sigma) = \begin{cases} T, & \text{if there exists } \mu \text{ such that } T(p)(\sigma\{\mu/x\}) = T \\ F, & \text{otherwise.} \end{cases}$$

$$T(S_1{=}S_2)(\sigma) = equal(M(S_1)(\sigma), M(S_2)(\sigma)) \quad (\text{here } equal\colon \Sigma\times\Sigma \to \{T,F\}).$$

(It should be noted that the p's are assertions *about* programs, and not themselves programming constructs. E.g., a boolean procedure bp with the declaration (in ALGOL 60 notation) <u>boolean</u> <u>procedure</u> bp; <u>begin</u> S; bp:= <u>true</u> <u>end</u>, will result in an infinite computation when called in a state σ for which S does not terminate, whereas $T(S;\underline{\text{true}})(\sigma)$ yields F.)

Next, we introduce the following abbreviations:

$p \vee q$	$\equiv (\neg p) \supset q$
$p \wedge q$	$\equiv \neg(p \supset \neg q)$
$p = q$	$\equiv (p\supset q)\wedge(q\supset p)$
$\underline{\text{if}}\ p\ \underline{\text{then}}\ q_1\ \underline{\text{else}}\ q_2\ \underline{\text{fi}}$	$\equiv (p\wedge q_1)\vee(\neg p\wedge q_2)$
$S \to p$	$\equiv (S;\underline{\text{true}}) \supset (S;p)$
$\{p\}S\{q\}$	$\equiv p \supset (S\to q)$
$[p]S[q]$	$\equiv p \supset (S;q)$.

(Below we apply the usual conventions on the priority of the logical operators $\neg,\wedge,\vee,\supset,=$.)

Let us now see what we obtain from these definitions in the last two cases: For each σ

$$T(\{p\}S\{q\})(\sigma) = \begin{cases} T, & \text{if, for all } \sigma', \text{ whenever } T(p)(\sigma) = T \text{ and } \sigma' = M(S)(\sigma), \text{ then } T(q)(\sigma') = T \\ F, & \text{otherwise.} \end{cases}$$

$$T([p]S[q])(\sigma) = \begin{cases} T, & \text{if, whenever } T(p)(\sigma) = T, \text{ then there exists } \sigma' \text{ such that } \sigma' = M(S)(\sigma) \text{ and } T(q)(\sigma') = T \\ F, & \text{otherwise.} \end{cases}$$

Thus, we encounter here the usual notions of *partial correctness* (in the formulation of Hoare [14]) and *total correctness*, see e.g., Manna. [20] Let us moreover point out that the meaning of our construct S;p (also appearing in Mirkowska & Salwicki [23]) is nothing but Dijkstra's weakest precondition wp(S,p) (provided that we restrict ourselves - as we do here- to deterministic programs; the nondeterministic case is investigated e.g. in De Bakker [6] and De Roever [27]).

A formula is called *valid* if, for *all* σ, $T(p)(\sigma) = T$, or $T(S_1{=}S_2)(\sigma) = T$, respectively. Examples of valid assertions are

$$S;\underline{\text{false}} = \underline{\text{false}} \tag{3.1}$$
$$S;(p\wedge q) = (S;p) \wedge (S;q) \tag{3.2}$$
$$S;(p\vee q) = (S;p) \vee (S;q) \tag{3.3}$$

Using p[s/x] to denote the result of replacing all occurrences of x in p by s, we also have the validity of

$$(x{:=}s);p = p[s/x] \tag{3.4}$$

provided that p *contains no subexpressions of the form* S;p'

$$(S_1;S_2);p = S_1;(S_2;p) \tag{3.5}$$
$$\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}};p = \underline{\text{if}}\ b\ \underline{\text{then}}\ S_1;p\ \underline{\text{else}}\ S_2;p\ \underline{\text{fi}} \tag{3.6}$$

Valid assertions expressing partial correctness are

$$\{p[s/x]\}\ x{:=}s\ \{p\} \tag{3.7}$$

provided that p *contains no subexpressions of the form* S;p'

$$\{p\}\ x{:=}s\ \{\exists y[p[y/x] \wedge x{=}s[y/x]]\} \quad \text{(Floyd [13])} \tag{3.8}$$
$$\{p\wedge b\}S_1\{r\} \wedge \{p\wedge\neg b\}S_2\{r\} \supset \{p\}\ \underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}}\ \{r\} \tag{3.9}$$

As examples of valid equivalences we mention

$$\underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \underline{\text{od}} = \underline{\text{if}}\ b\ \underline{\text{then}}\ S;\ \underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \underline{\text{od}}\ \underline{\text{else}}\ \Delta\ \underline{\text{fi}} \tag{3.10}$$
$$\underline{\text{if}}\ b\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \underline{\text{fi}};S = \underline{\text{if}}\ b\ \underline{\text{then}}\ S_1;S\ \underline{\text{else}}\ S_2;S\ \underline{\text{fi}} \tag{3.11}$$

A *deduction* is a construct of the form $\frac{\pi_1}{\pi_2}$, where π_1 and π_2 are formulae. In the formal proof theory, it will serve as a means for deriving new theorems from old ones (which are either axioms or previously derived theorems). Therefore, we are interested in the notion of a *sound* deduction: A deduction is called sound if the validity of its premise (π_1) implies validity of its conclusion (π_2). Examples of sound deductions are

$$\frac{p}{p[y/x]}\ , \text{ provided that y does not occur free in p} \tag{3.12}$$

$$\frac{\{p\}S_1\{q\} \wedge \{q\}S_2\{r\}}{\{p\}S_1;S_2\{r\}} \tag{3.13}$$

$$\frac{\{p\wedge b\}S\{p\}}{\{p\}\ \underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \underline{\text{od}}\ \{p\wedge\neg b\}} \tag{3.14}$$

$$\frac{p\supset q}{S;p\supset S;q} \qquad \frac{S_1 = S_2}{S;S_1 = S;S_2} \qquad \frac{S_1 = S_2}{S_1;p = S_2;p} \tag{3.15a,b,c}$$

An example of an invalid assertion is: $(p\supset q) \supset ((S;p) \supset (S;q))$. An unsound deduction is the following

$$\frac{\{\underline{\text{true}}\}x{:=}1;y{:=}2\{x{=}1\wedge y{=}2\}}{\{\underline{\text{true}}\}y{:=}1;y{:=}2\{y{=}1\wedge y{=}2\}} \ .$$

In a proof theory one selects certain valid formulae as axioms, and sound deductions as proof rules. For example, in Hoare's proof theory we encounter assertion (3.7) as an axiom, and assertion (3.9) and deductions (3.13) and (3.14) as proof rules, whereas in Dijkstra's system we find (3.1-3.6) and (3.15a). One then hopes to be able to derive a class of interesting program properties on the basis of these axioms and rules. The development of a formal proof theory is in particular motivated by two considerations:

- a judicious selection of axioms and rules may lead to a system which is *complete* for a certain class of properties - thus enabling

the programmer in that case to base all his proofs on the selected axioms and rules, without any appeal to facts outside the formal theory. (E.g., Hoare's system is incomplete, since the equivalence (3.10) is not derivable in it (see [4]). Addition of (3.10) yields a theory which fully characterizes the while statement in the same sense as investigated in a much more general setting in De Bakker & Meertens. [9]) Moreover, an appropriate choice of the axioms and rules may sometimes lead to a natural (implicit) definition of the meaning of the concepts concerned.
- Any system for computer verification of program correctness has to rely on some formalized proof theory which informs the computer as to what are the legal inferences of the system.

4. APPLICATIONS AND EXTENSIONS

In this section we present three case studies which illustrate the interface between semantics and proof theory. They are concerned with
- assignment to a subscripted variable
- weakest preconditions and the while statement
- parameter mechanisms for procedures.

In each case we hope to shed some light on a point which, simple as it may be, seems to be not yet fully understood in the literature.

4.1 Assignment to a subscripted variable

Consider the assignment statement x:=1. Clearly, {true}x:=1{x=1} is a desirable property of it, which is easily seen to be both valid, and derivable by Hoare's assignment axiom. Indeed, (x=1)[1/x] reduces to 1 = 1, which is equivalent with true. Now let us assume that our language has been extended with subscripted variables. We first of all have to give the semantics of this extension. This is rather straightforward, and omitted here (see [8]). What to do, however, with the proof theory? First we try to treat a subscripted variable a[s] in the same manner as a simple variable, allowing us to derive, e.g., {true}a[2]:=1{a[2]=1} (since true is equivalent with (a[2]=1)[1/a[2]]). Similarly we would then obtain

$$\{\underline{true}\}a[a[2]]:=1\{a[a[2]]=1\}, \qquad (4.1)$$

(assuming that true is also equivalent with (a[a[2]] = 1) [1/a[a[2]]]) but this formula can be shown to be *invalid* in the following way: It is not difficult to verify the validity of

$$\{a[1]=2 \wedge a[2]=2\}\ a[a[2]]:=1\ \{a[a[2]]=2\}. \qquad (4.2)$$

Since, obviously, $a[1]=2 \wedge a[2]=2 \supset$ true is valid, from (4.1) we obtain

$$\{a[1]=2 \wedge a[2]=2\}\ a[a[2]]:=1\ \{a[a[2]]=1\}$$

contradicting (4.2).

The solution to the invalidity of Hoare's axiom, when carried over directly to the subscripted variable case, is provided by refining the definition of substitution p[t/v], where v now ranges over both simple variables x and subscripted variables a[s]. By obvious reductions such as
$(p_1 \supset p_2)[t/v] \equiv p_1[t/v] \supset p_2[t/v]$, or
$(s_1=s_2)[t/v] \equiv (s_1[t/v] = (s_2[t/v])$, we arrive at the treatment of w[t/v], for v,w arbitrary variables. The cases where w and/or v are simple variables are rather straightforward and omitted here. [8] The heart of the definition consists of

$$b[s'][t/a[s]] \overset{df.}{\equiv} b[s'[t/a[s]]] \qquad (a \not\equiv b)$$

$$a[s'][t/a[s]] \overset{df.}{\equiv} \underline{if}\ s'[t/a[s]] = s\ \underline{then}\ t\ \underline{else}\ a[s'[t/a[s]]]\underline{fi}.$$

It can be shown that (3.7), taken with the new substitution definition, is valid. [8]

Example. (a[a[2]]= 1)[1/a[a[2]]] ≡ (if a[2][1/a[a[2]]] = a[2] then 1 else a[a[2][1/a[a[2]]]] fi = 1). By a few (omitted) simplifications, we reduce this to: if a[2] = 2 then a[1] = 1 else true fi. Thus, we obtain as instance of (3.7):

$$\{\underline{if}\ a[2] = 2\ \underline{then}\ a[1] = 1\ \underline{else}\ \underline{true}\ \underline{fi}\}\ a[a[2]] := 1\{a[a[2]] = 1\},$$

thus correcting (4.1).

4.2 Weakest preconditions and the while statement

Let us consider Theorem 4 of [12]. When stripped to its essentials (the presence of nondeterminacy is irrelevant here), the theorem can be phrased in our notation in the following way:

$$\frac{p \wedge b \supset S;p}{p \wedge (\underline{while}\ b\ \underline{do}\ S\ \underline{od};\underline{true}) \supset (\underline{while}\ b\ \underline{do}\ S\ \underline{od};(p \wedge \neg b))} \qquad (4.3)$$

It will be shown that this is nothing but a weaker version of (3.14) (this was first noted in [5]).

Assume (3.14) and the premise $p \wedge b \supset S;p$. We show that the conclusion of (4.3) is then derivable: Since $p \wedge b \supset S;p$, clearly, also $p \wedge b \wedge (S;\underline{true}) \supset S;p$, or, by simple propositional logic, $p \wedge b \supset (S;\underline{true} \supset S;p)$, i.e., $p \wedge b \supset (S \to p)$, or, in the partial correctness notation $\{p \wedge b\}S\{p\}$. Thus, the premise of (3.14) holds, and we infer the conclusion of (3.14): $\{p\}$ while b do S od $\{p \wedge \neg b\}$, which, in the same way, can be shown to be nothing but an abbreviation for the conclusion of (4.3). □

We here observe the advantages of an approach in which it is possible to formally compare notions such as partial correctness and weakest preconditions, thus clarifying the relationship between the various techniques.

4.3 Parameter mechanisms

By way of example we consider the parameter mechanisms of call-by-value and call-by-variable as occurring in the programming language PASCAL (this subsection is based on [1,2]). We extend the syntax as given in section 2 by introducing a class of procedure variables P, together with the constructs of procedure declaration and call. For the sake of simplifying the presentation here, we assume some restrictions: We have one procedure declaration P ⇐ <val x, var y | S>, where to the right of "⇐" we find a construct which has a formal value parameter x, a formal variable parameter y, and body S. A procedure call has the form P(t,v) with as actual parameters the integer expression t (for the formal x) and variable v (for the formal y).

We now outline how to provide a meaning to P(t,v) in the non-recursive case (no occurrences of P in S). For this purpose we first of all need the construct of a *block*: begin new z;S end, where z is any simple variable and S any statement. We assume that the reader has an intuitive understanding of this concept, and omit formal specification of its semantics (and corresponding proof rule). For this we refer for example to [1,2,15]. We also omit the precise definition of substitution in a statement, written as S[v/x], apart from mentioning that the new z... construct has the same variable binding effect as ∀z ... or ∫ ... dz has elsewhere in mathematics. Assuming these definitions, we introduce the following notation:

$\langle \underline{\text{val}}\ x,\ \underline{\text{var}}\ y \mid S\rangle\ (t,z) \overset{\text{df.}}{\equiv}$

$\quad \underline{\text{begin}}\ \underline{\text{new}}\ u;\ u:=t; S[u/x][z/y]\ \underline{\text{end}}$

$\langle \underline{\text{val}}\ x,\ \underline{\text{var}}\ y \mid S\rangle\ (t,a[s]) \overset{\text{df.}}{\equiv}$

$\quad \underline{\text{begin}}\ \underline{\text{new}}\ u_1,u_2; u_1:=t; u_2:=s;$

$\qquad S[u_1/x][a[u_2]/y]\ \underline{\text{end}}.$

Writing B as shorthand for <val x, var y | S>, we can now give concise rules for meaning and proofs for a procedure call P(t,v). Assume the declaration $P \Leftarrow B$. Then, for all σ,

$$M(P(t,v))(\sigma) = M(B(t,v))(\sigma),$$

and in the proof theory we might incorporate, for example

$$P(t,v) = B(t,v)$$

$$P(t,v);p = B(t,v);p,$$

or

$$\frac{\{p\}\ B(t,v)\ \{q\}}{\{p\}\ P(t,v)\ \{q\}}$$

depending on whether this proof theory favors equivalences, weakest preconditions, or a partial correctness approach.

Various approaches in the literature (e.g., [15,16]) tend to confuse procedure calls with substitution. Let us give an example of this: Consider the declaration $P_1 \Leftarrow \langle \underline{\text{var}}\ y1,y2 \mid y1:=2; y2:=3\rangle$ (with a slight deviation from our previous syntactic convention). The treatment of procedure calls as proposed in [16] would, through inappropriate use of substitution, result in deductions such as

$$\frac{\{\underline{\text{true}}\}\ y1:=2;\quad y2:=3\quad \{y1=2 \wedge y2=3\}}{\{\underline{\text{true}}\}\quad P_1(z,z)\quad \{z=2 \wedge z=3\}},$$

and, rightly considering this undesirable, its authors remedy this by forbidding calls as $P_1(z,z)$. We find our definition advantageous, since there is no contradiction in the inference

$$\frac{\{\underline{\text{true}}\}\quad \langle \underline{\text{var}}\ y1,y2 \mid y1:=2;\ y2:=3\rangle\ (z,z)\ \{z=3\}}{\{\underline{\text{true}}\}\qquad P_1(z,z)\qquad \{z=3\}}$$

because, by the B(t,v) definition, this reduces to the sound deduction

$$\frac{\{\underline{\text{true}}\}\ z:=2;\ z:=3\ \{z=3\}}{\{\underline{\text{true}}\}\quad P_1(z,z)\quad \{z=3\}}.$$

Remark. Observe that from (3.12) we deduce that

$$\frac{\{p\}S\{q\}}{\{p[y/x]\}S[y/x]\{q[y/x]\}}\text{, provided that } y \text{ does not occur free in } p,S \text{ or } q \quad (4.4)$$

is a sound proof rule. However, this rule does *not* allow the deduction

$$\frac{\{\underline{\text{true}}\}\ y1:=2;\ y2:=3\ \{y1=2 \wedge y2=3\}}{\{\underline{\text{true}}\}\quad z:=2;\quad z:=3\ \{\ z=2 \wedge\ z=3\}},$$

since the proviso of (4.4) is violated after substitution of z for either y1 or y2.

5. CONCLUSIONS

We have illustrated the connections between semantics and the foundations of program proving by an analysis of a few basic programming concepts and a fragment of the associated proof theory. We are convinced that the development of firm foundations for program proving has to rely heavily on a thorough study of the semantics of the concepts concerned, together with a careful application of it in the justification of the proof theory. There is currently a vigorous activity in this area, and our paper has touched only on a modest selection of the work in progress. For example, we have omitted all treatment of the investigations dealing with concepts such as recursion, nondeterministic and parallel programming, or (abstract) data types. *Recursion* is well-understood both as to its semantics, where the so-called least fixed point characterization is used (described for example in [4]), and as to its proof theory, which centers around an induction rule due to Scott. [28] (It may be of some interest to mention here that the discovery of this rule formed part of the motivation for Scott's recent Turing award.) Certain doubts shed on the validity of the least fixed point approach in the presence of, for example the call-by-value parameter mechanism, were clarified in our [7]. For *parallel* programming, we have good hopes for the development of appropriate semantics on the basis of the mathematical constructions of Plotkin [25] and Smyth. [30] We consider it an interesting challenge for future work to justify the proof theory as proposed for example in Owicki & Gries [24] on the basis of these semantics. As to the study of abstract data types, we feel that it is as yet too early to single out any definitive developments in this field.

By way of conclusion, let us recall the aims of a mathematical theory of programming languages as stated in the introduction, namely an improved insight into the fundamental programming concepts, and application of this in the methodology of program design and verification. When we compare the present situation with that of say ten years ago (cf. [3]), we may well be proud of the achievements in semantics during this period. Though still in a state of intense development, there are now some major results and techniques in semantics which are here to stay, allowing the programmer a better understanding of his most precious tool.

APPENDIX

In this appendix we give the semantics of the while statement, and present a new type of assertion which provides an alternative to the $\underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \underline{\text{od}};p$ construct.

Let us assume the usual partial ordering on the elements ϕ,ϕ' in $\Sigma \xrightarrow[\text{part}]{} \Sigma$ ($\phi \subseteq \phi'$ if, for all σ, either $\phi(\sigma)$ is undefined, or $\phi(\sigma)$ and $\phi'(\sigma)$ are both defined and yield the same value). Let, for a chain $\phi_0 \subseteq \phi_1 \subseteq \cdots \subseteq \phi_i \subseteq \ldots$, $\bigcup_{i=0}^{\infty} \phi_i$ denote its least upper bound. We put

$$M(\underline{\text{while}}\ b\ \underline{\text{do}}\ S\ \underline{\text{od}}) = \bigcup_{i=0}^{\infty} \phi_i$$

where, for each σ,

$$\phi_0(\sigma) \;=\; \text{undefined}$$

$$\phi_{i+1}(\sigma) = \begin{cases} \phi_i(M(S)(\sigma)), & \text{if } T(b)(\sigma) = T \\ \sigma, & \text{if } T(b)(\sigma) = F. \end{cases}$$

Furthermore, let us extend the definition of the class of assertions with the clause

$$p ::= \ldots \mid \underline{\text{rep}}\ b;S\ \underline{\text{per}}\ p$$

for which we define the function T in the following manner: For each $\gamma,\gamma' \in \Sigma \to \{T,F\}$, we put $\gamma \subseteq \gamma'$ if, for each σ, $\gamma(\sigma) \Rightarrow \gamma'(\sigma)$. Again, $\bigcup_{i=0}^{\infty} \gamma_i$ denotes the lub of the chain $\gamma_0 \subseteq \gamma_1 \subseteq \cdots \subseteq \gamma_i \subseteq \ldots$. We now put

$$T(\underline{rep}\ b;S\ \underline{per}\ p) = \bigcup_{i=0}^{\infty} \gamma_i$$

where, for each σ,

$$\gamma_0(\sigma) = F$$

$$\gamma_{i+1}(\sigma) = \begin{cases} \gamma_i(M(S)(\sigma)), & \text{if } T(b)(\sigma) = T \\ T(p)(\sigma), & \text{if } T(b)(\sigma) = F. \end{cases}$$

On the basis of these definitions we can then show the validity of assertions such as

$$\underline{while}\ b\ \underline{do}\ S\ \underline{od};p = \underline{rep}\ b;S\ \underline{per}\ p \quad (A.1)$$

$$\underline{rep}\ b;S\ \underline{per}\ p = \underline{if}\ b\ \underline{then}\ S;\underline{rep}\ b;S\ \underline{per}\ p\ \underline{else}\ p\ \underline{fi} \quad (A.2)$$

and the soundness of a deduction such as

$$\frac{q = \underline{if}\ b\ \underline{then}\ S;q\ \underline{else}\ p\ \underline{fi}}{\underline{rep}\ b;S\ \underline{per}\ p \supset q} \quad (A.3)$$

(Observe that (A.1 - A.3) together yield a least-fixed-point characterization of while b do S od;p. Cf. De Bakker & De Roever, [10] p. 187.)

REFERENCES

[1] K.R. Apt & J.W. de Bakker, Exercises in denotational semantics, in Proc. 5th Symp. Mathematical Foundations of Computer Science (A. Mazurkiewicz, ed.), 1-11, Lecture Notes in Computer Science 45, Springer, 1976.
[2] K.R. Apt & J.W. de Bakker, Semantics and proof theory of PASCAL procedures, in Proc. 4th Coll. Automata, Languages and Programming, Lecture Notes in Computer Science, Springer, 1977.
[3] J.W. de Bakker, Semantics of programming languages, in Advances in Information Systems Science (J.T. Tou, ed.), Vol. 2, Plenum Press, 1969, 173-227.
[4] J.W. de Bakker, The fixed point approach in semantics: theory and applications, in Foundations of Computer Science (J.W. de Bakker, ed.), 3-53, Mathematical Centre Tracts 63, 1975.
[5] J.W. de Bakker, Flow of control in the proof theory of structured programming, in Proc. 16th IEEE Symp. Foundations of Computer Science, Berkeley, 1975, 29-33.
[6] J.W. de Bakker, Semantics and termination of non-deterministic recursive programs, in Proc 3d Coll. Automata, Languages and Programming (S. Michaelson & R. Milner, eds), Edinburgh University Press, 1976, 435-477.
[7] J.W. de Bakker, Least fixed points revisited, Theoretical Computer Science, 2, 1976, 155-181.
[8] J.W. de Bakker, Correctness proofs for assignment statements, Report IW 55/76, Mathematical Centre, 1976.
[9] J.W. de Bakker & L.G.L.T. Meertens, On the completeness of the inductive assertion method, Journal of Computer and System Sciences, 11, 1975, 323-357.
[10] J.W. de Bakker & W.P. de Roever, A calculus for recursive program schemes, in Proc. 1st Coll. Automata, Languages and Programming (M. Nivat, ed.), North-Holland, 1973, 167-196.
[11] J.E. Donahue, Complementary Definitions of Programming Language Semantics, Lecture Notes in Computer Science 42, Springer, 1976.
[12] E.W. Dijkstra, Guarded commands, nondeterminacy and formal derivation of programs, Comm. ACM, 18, 1975, 453-457.
[13] R.W. Floyd, Assigning meanings to programs, in Proc. Symp. in Applied Mathematics, vol. 19 - Math. Aspects of Computer Science (J.T. Schwartz, ed.), AMS 1967, 19-32.
[14] C.A.R. Hoare, An axiomatic basis for computer programming, Comm. ACM, 12, 1969, 576-580.
[15] C.A.R. Hoare, Procedures and parameters, an axiomatic approach, in Symp. on Semantics Algorithmic Languages (E. Engeler, ed.), 102-116, Lecture Notes in Mathematics 188, Springer, 1971.
[16] S. Igarashi, R.L. London & D.C. Luckham, Automatic program verification I: A logical basis and its implementation, Acta Informatica 4, 1975, 145-182.
[17] G. Ligler, A mathematical approach to language design, Conf. Record Second ACM Symp. on Principles of Programming Languages, Palo ALto, 1975, 41-53.
[18] G. Ligler, Surface properties of programming language constructs, Int. Symp. on Proving and Improving Programs, Arc-et-Senans, IRIA 1975, 299-323.
[19] P. Lucas & K. Walk, On the formal description of PL/I, Annual Review in Automatic Programming, 6, 1969, 105-182.
[20] Z. Manna, Mathematical Theory of Computation, McGraw-Hill, 1974.
[21] R. Milne & C. Strachey, A Theory of Programming Language Semantics, Chapman & Hall, 1976.
[22] R. Milner, Program semantics and mechanized proof, in Foundations of Computer Science II (K.R. Apt & J.W. de Bakker, eds), Mathematical Centre Tracts 82, 1976.
[23] G. Mirkowska & A. Salwicki, A complete axiomatic characterization of algorithmic properties of block-structures programs with procedures, in Proc. 5th Symp. Mathematical Foundations of Computer Science (A. Mazurkiewicz, ed.), 602-606. Lecture Notes in Computer Science 45, Springer, 1976.
[24] S. Owicki & D. Gries, Verifying properties of parallel programs: an axiomatic approach, Comm. ACM, 19, 1976, 279-285.
[25] G. Plotkin, A powerdomain construction, SIAM J. on Computing, 5, 1976, 452-487.
[26] V.R. Pratt, Semantical considerations on Floyd-Hoare logic, Proc. 17th IEEE Symp. on Foundations of Computer Science, Houston, 1976
[27] W.P. de Roever, Dijkstra's predicate transformer, nondeterminism, recursion and termination, in Proc. 5th Symp. Mathematical Foundations of Computer Science (A. Mazurkiewicz, ed.), 472-481, Lecture Notes in Computer Science 45, Springer, 1976.
[28] D. Scott & J.W. de Bakker, A theory of programs, unpublished memo, 1969.
[29] D. Scott & C. Strachey, Towards a mathematical semantics for computer languages, in Proc. Symp. Computers and Automata (J. Fox, ed.), Polytechnic Institute of Brooklyn, 1971, 19-46.
[30] M.B. Smyth, Powerdomains, in Proc. 5th Symp. Mathematical Foundations of Computer Science (A. Mazurkiewicz, ed.), 537-543. Lecture Notes in Computer Science 45, Springer, 1976.
[31] R.D. Tennent, The denotational semantics of programming languages, Comm. ACM, 19, 1976, 437-453.
[32] A. van Wijngaarden et al. (eds), Revised Report on the Algorithmic Language ALGOL 68, Mathematical Centre Tracts, 50, 1976.