Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

45

Mathematical Foundations of Computer Science 1976

Springer-Verlag
Berlin · Heidelberg · New York

TP391
X76
M47
TP301-53
M426
1976
(2)

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

45

Mathematical Foundations of Computer Science 1976

Proceedings, 5th Symposium,
Gdańsk, September 6–10, 1976

Edited by A. Mazurkiewicz

TP
L
V.45

Springer-Verlag
Berlin · Heidelberg · New York 1976

Editor
Antoni Mazurkiewicz
Computation Centre
Polish Academy of Sciences
P.O.Box 22,
00-901 Warszawa/Poland

AMS Subject Classifications (1970): 02C99, 02E10, 02E15, 02H10, 18B20, 68A05, 68A10, 68A20, 68A25, 68A30, 68A45, 94A25, 94A30

ISBN 3-540-07854-1 Springer-Verlag Berlin · Heidelberg · New York
ISBN 0-387-07854-1 Springer-Verlag New York · Heidelberg · Berlin

Printed in Germany
Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.

MFCS'76

FOREWORD

This volume contains papers which were contributed for presentation at the Symposium on Mathematical Foundations of Computer Science — MFCS'76, held in Gdańsk, Poland, September 6-10, 1976. This symposium is the 5th in the series of annual MFCS symposia organized in turn in Poland /every even year/ and Czechoslovakia /every odd year/ . The aim of these symposia is to promote and to develop the mathematical approach to the basic computational phenomena.

The articles in these Proceedings consist of a number of invited papers and short communications concerning mathematical results motivated by practical problems and related to:

- Programs and computations,
- Programming languages,
- Data bases and information retrieval systems,
- Analysis and complexity of algorithms,
- Formal languages and automata.

The scientific interest in the above topics is increasing rapidly; an example of this interest can be seen in the number of papers submitted for the Symposium. The Program Committee has been forced to reject more than half of them /sometimes valuable ones/. The main guideline for selecting papers was their orginality and relevance to the subject of the Symposium.

The Symposium is being organized by the Computation Centre of the Polish Academy of Sciences in cooperation with the University of Gdańsk.

The organizers of the Symposium are grateful to all authors for their valuable contributions and to all people who helped in the organization of the Symposium. The main part of the organizational work has been done by the following members of the committee: E.Czuchajew , P.Dembiński /Vice-Chairman/, C.Góral, W.Kwasowiec, J.Leszczyłowski , W.Lipski,Jr. , A.Mazurkiewicz /Symposium Chairman/, A.W.Mostowski , B.Rykaczewska, J.Winkowski /Program Chairman/. The organizers are specially indebted to J.Winkowski, who has taken the greatest part in the preparation of this volume.

The help of Springer-Verlag, which has produced these Proceedings is highly appreciated.

Antoni Mazurkiewicz

Warsaw, May 1976

CONTENTS

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 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 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:

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

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],...), 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

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

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 $w \in V$.
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 $S \in \mathcal{S}$, 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 $x \in SV$, $\varepsilon(x) \in A$ may be defined, and for each $a \in AV$ and $\nu \in I$, $\varepsilon(a,\nu)$ *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