ALGOL 68 as a metalanguage for denotational semantics†

F. G. Pagan*

Computer Science Group, Memorial University of Newfoundland, St. John's, Canada A1C 5S7

The possibility of using ALGOL 68 as a metalanguage for 'denotational' definitions of the semantics of programming languages is considered, using the simple language LOOP as an example. The approach is found to be a viable one if the 'partial parametrisation' feature is added to ALGOL 68. The advantages of using a general purpose programming language as a definitional medium are briefly discussed.

(Received September 1977)

1. Background

The *denotational* approach (also known as the *mathematical* or Scott/Strachey approach) to the formal semantic specification of programming languages is currently receiving much attention. The reader is referred to Tennent (1976) for an introduction to this approach and for an extensive bibliography. The notion that program semantics can be wholly described in terms of mathematical functions plays a central role in the method, which is more abstract than the *operational* or *constructive* methods but less abstract than the purely *axiomatic* or *implicit* approaches (Hoare and Lauer, 1974). Leaving aside the last mentioned approaches, formal definitions of languages often take the form of *abstract interpreters* of various sorts (Reynolds, 1972).

Many workers have taken the position that formal definitions should be expressed in a metalanguage, possibly an actual programming language itself, suitable for computer processing; for example Anderson, Belz and Blum (1976) and Pagan (1976) take this view in the context of operational (implementation oriented) definitions. In the context of denotational definitions, Mosses (1975) proposes a specially designed, machine readable metalanguage, and Mosses (1976) discusses its possible utilisation in a compiler generation system.

The present paper may be viewed as a sequel to Pagan (1976) in which the author discussed the use of a general purpose programming language, in particular ALGOL 68, as a metalanguage for operational semantic definitions and concluded that it was at least as well suited for this purpose as the purely formal Vienna Definition Language. Here we consider some similar ideas in the context of denotational definitions. The principal points will be introduced in connection with the trivial programming language LOOP, which is also the first example considered in Tennent (1976). It should perhaps be pointed out that, in advocating a general purpose programming language as a notational medium, the paper is not concerned with the mathematical or utilitarian adequacy of the denotational approach and is neutral with respect to the question of its superiority (or otherwise) over other approaches.

2. LOOP and its denotational semantics

LOOP is an extremely simple language concerned with non-negative integer values, its main features being a successor operator, an assignment statement, and a simple loop construct. An informal understanding of LOOP may be readily gained from the following BNF grammar:

⟨program⟩ ::= **read** ⟨variable⟩; ⟨command list⟩;
write ⟨expression⟩
⟨command list⟩ ::= ⟨command⟩ | ⟨command list⟩;
⟨command⟩
⟨command⟩ ::= ⟨variable⟩ := ⟨expression⟩
| **to** ⟨expression⟩ **do** ⟨command⟩
| (⟨command list⟩)
⟨expression⟩ ::= 0 | ⟨variable⟩ | **succ** ⟨expression⟩

Although the language is obviously of no practical value, it will suffice for illustrating the points made in this paper. The sample program

read x; $y := x$;
to x **do** $y :=$ **succ** y;
write succ succ y

corresponds to the input/output function

$$f(x) = 2x + 2$$

or, in lambda notation,

$$\lambda x \,.\, 2x + 2$$

or, as an ALGOL 68 routine text,

$$(\textbf{int } x) : 2 \times x + 2$$

The denotational definition of the semantics of LOOP as given in Tennent (1976) is summarised in **Fig. 1.** The first part specifies the various syntactic domains and corresponding metavariables; for example,

$$E : \textbf{Exp}$$

specifies that **Exp** (corresponding to the set of all expressions in LOOP) is a syntactic domain and that E (and possibly E', E_1, E_2, etc.) will be used as a metavariable over this domain. The relationships among the syntactic domains are given in terms of production rules which are a mild abstraction from the BNF rules. The form of these productions is convenient for the subsequent semantic specifications. Taking a deeper view of the notion of abstract syntax, however, we could have eliminated all non-essential 'syntactic sugar' and specified the domain relationships by writing the following equations

$$\textbf{Prog} = \textbf{Var} \times \textbf{Cmd} \times \textbf{Exp}$$
$$\textbf{Cmd} = (\textbf{Cmd} \times \textbf{Cmd}) + (\textbf{Var} \times \textbf{Exp}) + (\textbf{Exp} \times \textbf{Cmd})$$
$$\textbf{Exp} = \{0\} + \textbf{Var} + (\{\text{succ}\} \times \textbf{Exp})$$

The only semantic domains specified in Fig. 1 are the set of numbers (non-negative integers) **N** and the set of states **S**, where a particular state σ is a function mapping variables into numbers. The functions $\mathscr{M}$, $\mathscr{C}$, and $\mathscr{E}$ define the meaning of the various language constructs in terms of functions involving the semantic domains. Thus, according to the 'semantic functions' section of Fig. 1, the meaning of a program is a function from numbers to numbers, the meaning of a command is a function from states to states, and the meaning of an expression is a function from states to numbers. The semantic equations give the detailed definitions of the function-producing functions $\mathscr{M}$, $\mathscr{C}$, and $\mathscr{E}$ with the aid of the following notational devices:

*Now at Department of Computer Science, Southern Illinois University, Carbondale, Illinois 62901, USA.
†This work has been supported by the National Research Council of Canada, grant number A3605.

Syntactic domains

Ψ: **Prog**	(programs)
Γ: **Cmd**	(commands)
E: **Exp**	(expressions)
Ξ: **Var**	(variables)

Abstract productions

Ψ ::= **read** Ξ; Γ; **write** E
Γ ::= Γ_1; Γ_2 | Ξ := E | **to** E **do** Γ | (Γ)
E ::= 0 | Ξ | **succ** E

Semantic domains

v: **N**	(non-negative integers)
σ: **S** = **Var** → **N**	(states)

Semantic functions

$\mathscr{M}$: **Prog** → **N** → **N**
$\mathscr{C}$: **Cmd** → **S** → **S**
$\mathscr{E}$: **Exp** → **S** → **N**

Semantic equations

$\mathscr{M}[\![$**read** Ξ; Γ; **write** $E]\!]\ v = \mathscr{E}[\![E]\!]\sigma_f$
where $\sigma_f = \mathscr{C}[\![\Gamma]\!]\ (\sigma_i[v/\Xi])$
where $\sigma_i[\![\Xi']\!] = 0$ for all Ξ': **Var**
$\mathscr{C}[\![(\Gamma)]\!] = \mathscr{C}[\![\Gamma]\!]$
$\mathscr{C}[\![\Gamma_1; \Gamma_2]\!] = \mathscr{C}[\![\Gamma_2]\!] \circ \mathscr{C}[\![\Gamma_1]\!]$
$\mathscr{C}[\![\Xi := E]\!]\sigma = \sigma[\mathscr{E}[\![E]\!]\sigma/\Xi]$
$\mathscr{C}[\![$**to** E **do** $\Gamma]\!]\ \sigma = ((\mathscr{C}[\![\Gamma]\!])^v)\sigma$
where $v = \mathscr{E}[\![E]\!]\sigma$
$\mathscr{E}[\![0]\!]\sigma = 0$
$\mathscr{E}[\![\Xi]\!]\sigma = \sigma[\![\Xi]\!]$
$\mathscr{E}[\![$**succ** $E]\!]\sigma = \mathscr{E}[\![E]\!]\sigma + 1$

Fig. 1 Original denotational definition of LOOP

```
proc m = (prog p) proc(nng)nng :
 (nng n) nng : (
  state initst = (var v) nng : 0;
  state finalst = c (body of P) (update (initst, readvar
   of P, n));
  e (writeexpr of P) (finalst) );

proc c = (cmd cm) proc(state)state :
 case cm in
  (seq se) : (state s) state :
          c (cmd2 of SE) (c (cmd1 of SE) (s)),
  (asst as) : (state s) state :
          update (s, lhs of AS, e (rhs of AS) (s)),
  (loop lp) : (state s) state :
          iter (c (body of LP), e (lim of LP) (s)) (s)
 esac;

proc e = (exp ex) proc(state)nng :
 case ex in
  (int) : (state s) nng : 0,
  (var v) : (state s) nng : s (V),
  (succ su) : (state s) nng : e (opd of SU) (s) + 1
 esac;

proc iter = (proc(state)state f, nng n) proc(state)state :
 if n = 0
 then (state s) state : s
 else proc(state)state g = iter (f, n - 1);
  (state s) state : F (G (s))
 fi;

proc update = (state s, var v, nng n) state :
 (var id) nng : (id = V | N | S (id))
```

Fig. 2 First attempt to specify semantic functions in ALGOL 68

1. Arguments from syntactic domains are enclosed by $[\![$ and $]\!]$.
2. Single symbol arguments from semantic domains are not parenthesised, e.g. $\mathscr{E}[\![0]\!]\sigma$ means $(\mathscr{E}[\![0]\!])\ (\sigma)$.
3. $\sigma[v/\Xi]$ denotes the state which is the same function as σ except that it maps the variable Ξ into the number v.
4. An argument α in an equation of the form $f\alpha = g\alpha$ may be 'cancelled', resulting in a functional equality $f = g$.
5. f^n denotes the function obtained by composing f with itself n times; if $n = 0$, it is the identity function.

A program Ψ with input v will produce as output the value of $\mathscr{M}[\![\Psi]\!]v$. Given a suitable mechanism for constructing and applying functional values, the denotational definition constitutes a kind of interpreter for the defined language. Such a mechanism would be provided by a high level programming language capable of manipulating functions as values and of structuring data in ways corresponding to the various operations on domains.

3. ALGOL 68 as metalanguage

The abstract syntax of LOOP can be expressed in ALGOL 68 as follows:

```
mode prog = struct (var readvar, cmd body, exp writeexpr),
  cmd = union (seq, asst, loop),
  exp = union (int #0#, var, succ),
  var = char,
  seq = struct (ref cmd cmd1, cmd2),
  asst = struct (var lhs, exp rhs),
  loop = struct (exp lim, ref cmd body),
  succ = struct (ref exp opd)
```

It is convenient to introduce **seq, asst, loop,** and **succ** as syntactic subdomains and to assume that LOOP variables consist of single characters. It is not difficult to write a readable ALGOL 68 program which will translate any LOOP source program into an internal value of mode **prog.** These aspects are discussed in more detail in Pagan (1976; 1977).

The semantic domains **N** and **S** are modelled by the following additional definitions:

```
mode nng = int #>=0#,
  state = proc(var)nng
```

It remains to code the meaning functions $\mathscr{M}$, $\mathscr{C}$, and $\mathscr{E}$ as ALGOL 68 procedures; we shall also need auxiliary procedures *iter*(f, n) corresponding to the notation f^n and *update*(s, v, n) corresponding to $s[n/v]$. **Fig. 2** shows a straightforward but (unfortunately) incorrect encoding of these procedures. Note the close correspondence with the original specifications and the way in which syntactic components are extracted by means of the field selectors defined in the mode definitions for the abstract syntax.

The procedures are incorrect because of scope violations in the routine texts specifying the functional values to be returned. The scope of an ALGOL 68 routine is restricted by the scopes of the nonlocal identifiers used in its body, so that a routine value cannot be passed out of a procedure if it makes reference to identifiers defined only within that procedure. The offending occurrences of identifiers are shown in upper case for emphasis in Fig. 2.

Our real need is for a single routine text to be capable of yielding different routines at different times when evaluated in different environments. Thus the routine text

(int x**) int** : $x + y$

would yield the 'incrementing' routine given by

(int x**) int** : $x + 1$

when evaluated in an environment where $y = 1$, and the 'decrementing' routine given by

```
proc m = (prog p) proc(nng)nng :
 ((PROG P, nng n) nng : (
  state initst = (var v) nng : 0;
  state finalst =
   c (body of p) (update (initst, readvar of p, n));
  e (writeexpr of p) (finalst) )) (P,);

proc c = (cmd cm) proc(state)state :
 case cm in
  (seq se) : ((SEQ SE, state s) state :
              c (cmd2 of se) (c (cmd1 of se) (s))) (SE,),
  (asst as) : ((ASST AS, state s) state :
              update (s, lhs of as, e (rhs of as) (s))) (AS,),
  (loop lp) : ((LOOP LP, state s) state :
              iter (c (body of lp), e (lim of lp) (s)) (s)
               ) (LP,)
 esac;

proc e = (exp ex) proc(state)nng :
 case ex in
  (int) : (state s) nng : 0,
  (var v) : ((VAR V, state s) nng : s (v)) (V,),
  (succ su) : ((SUCC SU, state s) nng :
              e (opd of su) (s) + 1) (SU,)
 esac;

proc iter = (proc(state)state f, nng n) proc(state)state :
 if n = 0
 then (state s) state : s
 else proc(state)state g = iter (f, n - 1);
  ((PROC(STATE)STATE F, G, state s) state : f (g (s))
   ) (F, G,)
 fi;

proc update = (state s, var v, nng n) state :
 ((STATE S, VAR V, id, NNG N) nng : (id = v | n | s (id))
  ) (S, V, , N)
```

Fig. 3 Specification of semantic functions in ALGOL 68 with partial parametrisation

(**int** x) **int** : $x - 1$

when evaluated in an environment where $y = -1$. Standard ALGOL 68 (van Wijngaarden, 1976) simply does not provide this capability, which is an important one for a 'higher order' programming language (Reynolds, 1972). Problems arising from this deficiency are not infrequent, and *partial parametrisation* (Lindsey, 1974; Lindsey, 1976) has been officially recommended as an appropriate ALGOL 68 superlanguage feature to alleviate them. With this feature, a call $f(1,)$ of the procedure defined by

proc f = (**int** y, x) **int** : $x + y$

yields the incrementing routine of mode **proc(int)int** as specified above and $f(-1,)$ yields the decrementing routine.

In **Fig. 3,** the procedures defining LOOP have been rewritten using partial parametrisation; here the use of upper case serves to highlight the differences from Fig. 2. Note that, although the various routine texts contain extra parameters, they are immediately partially parametrised with the appropriate nonlocal values so that the desired routine values are passed out. Given an implementation of an ALGOL 68 superlanguage incorporating partial parametrisation, the procedures of Fig. 3 provide an actual processor for abstract LOOP programs, so that the call $m(p)(n)$ will yield the number output by a program p with input n.

It is possible, but very inconvenient, to simulate partial parametrisation in standard ALGOL 68 with the aid of auxiliary structures where one of the fields is a routine and the remaining fields are some of the actual parameter values for the routine. This technique, which is also used in Rayward-Smith (1977) in a different problem context, arises from the concepts of 'defunctionalisation' and *closure* used in Reynolds (1972) and in earlier work. In the present case, it involves defining various unions of the auxiliary structures, writing explicit auxiliary procedures to apply the routine in a structure to its actual parameters and altering the main semantic procedures accordingly. For LOOP, three of the necessary auxiliary modes are given by

```
mode snfna = struct (proc(var, state)nng rout, var parm),
  snfnb = struct (proc(succ, state)nng rout, succ parm),
  snfn = union(proc(state)nng, snfna, snfnb)
```

and **state** must be redefined as

```
mode state = struct (proc(state, var, nng, var)nng rout,
                     ref state parm1, var parm2, nng parm3)
```

Then, given the additional procedures

```
proc apply snfn = (snfn fn, state s) nng:
 case fn in
  (proc(state)nng p) : p(s),
  (snfna sa) : (rout of sa) (parm of sa, s),
  (snfnb sb) : (rout of sb) (parm of sb, s)
 esac;
proc apply state = (state s, var v) nng :
 (rout of s) (parm1 of s, parm2 of s, parm3 of s, v)
```

the main semantic procedure e can be recoded as follows:

```
proc e = (exp ex) snfn :
 case ex in
  (int) : (state s) nng : 0,
  (var v) : snfna ((var v, state s) nng :
   apply state (s, v), v),
  (succ su) : snfnb ((succ su, state s) nng :
   apply snfn (e (opd of su), s) + 1, su)
 esac
```

The complete specifications, which are not shown here, are almost twice as long as those using partial parametrisation and are considerably more complex, ungainly and difficult to comprehend.

4. Some extensions, alternatives, and conclusions

Although the LOOP language is very small and its definition is correspondingly simple, there would appear to be no difficulty in generalising the techniques illustrated above to include the various other devices employed in denotational definitions. The ALGOL 68 mode system has sufficient power and generality to model closely complicated domains involving concepts such as environments, stores and continuations; for example, an equation of the form

$$\mathbf{A} = \mathbf{B} \to \mathbf{C}^* \to (\mathbf{D} \times \mathbf{E}) \to (\mathbf{F} + \mathbf{G})$$

might be modelled by a definition of the form

```
mode a = proc(b)proc(ref[ ]c)proc(struct(d d, e e)) union(f, g)
```

The operations of injection, inspection, and projection are provided by the uniting coercion and the conformity clause. Conditional functions, tuples and selections, lambda expressions, and so forth, all have direct analogues in ALGOL 68. ALGOL 68 is thus a viable metalanguage for large scale denotational definitions, at least if partial parametrisation is used.

One of the major advantages of using a programming language as the metalanguage of a definition is the resulting executability (or, at least, 'compilability') of the definition. (It may be that execution will be so inefficient that it is to all intents and purposes impossible.) This can greatly aid the

```
[abs "a" : abs "z"] nng stg;

proc int prog = (prog p, nng input) nng : (
 for i from abs "a" to abs "z" do stg[i] := 0 od;
 stg [abs readvar of p] := input;
 int cmd (body of p);
 int expr (writeexpr of p));

proc int cmd = (cmd cm) void :
 case cm in
  (seq se) : (int cmd (cmd1 of se); int cmd (cmd2 of se)),
  (asst as) : stg [abs lhs of as] := int expr (rhs of as),
  (loop lp) : to int expr (lim of lp) do int cmd (body of lp) od
 esac;

proc int expr = (exp ex) nng :
 case ex in
  (int) : 0,
  (var v) : stg [abs v],
  (succ su) : int expr (opd of su) + 1
 esac
```

Fig. 4 Operational definition in ALGOL 68

language definer in achieving a complete, consistent, and 'bug-free' specification, for language defining and programming are actually very similar activities in certain respects. Moreover, if the defining language is a 'mainstream' language like ALGOL 68 as opposed to a specially invented notation or a primitive scheme like pure LISP (where the lack of strong type checking is a serious disadvantage), then the definition will be more acceptable and understandable to the community of implementors and users. In this connection, it is important that the executable definition be comparable to or possibly better than a purely formal definition with respect to qualities such as readability and conciseness. In the author's opinion, this is achievable in the case of ALGOL 68 with partial parametrisation but not in the case of strict ALGOL 68 (and certainly not in the case of other 'mainstream' languages); it is to be hoped that future ALGOL 68 compilers will permit the use of this additional feature.

When viewed purely as an implementation of LOOP, the specifications in Fig. 3 are indeed strange, arcane and extremely inefficient, and the specifications using strict ALGOL 68 are even more so. The latter have been tested with the aid of an ALGOL 68 compiler; as expected, they will slowly but successfully interpret simple LOOP programs with small input values. For example, interpretation of the program given earlier for computing the function $\lambda x \,.\, 2x + 2$ with input value 2 involves about 70 function calls with a maximum recursion depth of 18. The lack of clarity and usefulness of Fig. 3 *as a processor* as opposed to a *definition* is not surprising, since denotational definitions are much less implementation oriented than operational definitions. If we had wanted the definition to constitute a clear and useful processor (probably at the expense of other uses such as proving correctness of programs), we should have taken an operational approach and written specifications such as those of **Fig. 4.**

References

ANDERSON, E. R., BELZ, F. C. and BLUM, E. K. (1976). SEMANOL (73) A Metalanguage for Programming the Semantics of Programming Languages, *Acta Informatica*, Vol. 6, pp. 109-131.

HOARE, C. A. R. and LAUER, P. E. (1974). Consistent and Complementary Formal Theories of the Semantics of Programming Languages, *Acta Informatica*, Vol. 3, pp. 135-153.

LINDSEY, C. H. (1974). Partial Parametrization, *ALGOL Bulletin*, No. 37, pp. 24-26.

LINDSEY, C. H. (1976). Specification of Partial Parametrization Proposal, *ALGOL Bulletin*, No. 39, pp. 6-9.

MOSSES, P. D. (1975). The Semantics of Semantic Equations, Proc. 3rd Symp. on Mathematical Foundations of Computer Science, Springer-Verlag Lecture Notes in Computer Science, No. 28, pp. 409-422.

MOSSES, P. D. (1976). Compiler Generation using Denotational Semantics, Proc. 5th Symp. on Mathematical Foundations of Computer Science, Springer-Verlag Lecture Notes in Computer Science, No. 45, pp. 436-441.

PAGAN, F. G. (1976). On Interpreter-Oriented Definitions of Programming Languages, *The Computer Journal*, Vol. 19, pp. 151-155.

PAGAN, F. G. (1977). ALGOL 68 as an Implementation Language for Portable Interpreters, Proc. Strathclyde ALGOL 68 Conf., *SIGPLAN Notices*, Vol. 12, No. 6, pp. 54-62.

RAYWARD-SMITH, V. J. (1977). Using Procedures in List Processing, Proc. Strathclyde ALGOL 68 Conf., *SIGPLAN Notices*, Vol. 12, No. 6, pp. 179-183.

REYNOLDS, J. C. (1972). Definitional Interpreters for Higher-Order Programming Languages, Proc. 25th ACM National Conf., pp. 717-740.

TENNENT, R. D. (1976). The Denotational Semantics of Programming Languages, *CACM*, Vol. 19, pp. 437-453.

VAN WIJNGAARDEN, A. *et al.* (1976). Revised Report on the Algorithmic Language ALGOL 68, Springer-Verlag, also in *Acta Informatica*, Vol. 5, Parts 1-3 (1975) and *SIGPLAN Notices*, Vol. 12, No. 5, pp. 1-70 (1977).

Book review

Structural Analysis and Design (Volumes 1 and 2), 1978. Infotech, £120)

The first volume is an analysis and bibliography of the processes of analysis and design. It takes the form of a series of extracts from many sources, including volume 2, linked by an editorial commentary. This format is very effective and it succeeds in demonstrating both the need for a methodology and also the schools of thought which lead to the competing methodologies. SADT, Michael Jackson Methodology, Warnier-Orr Methodology and Structured Design are each concisely and clearly described.

The second volume contains 19 invited papers, the first of which, by R. R. Brown, includes a productivity analysis of two application developments at Hughes Aircraft in 1974 and 1972/3: One of the rare published case studies. An elegant paper by S. N. Griffiths compares methodologies and finds much merit in Michael Jackson. It also takes an interesting perspective view of the current structured design scene together with some predictions for the future.

The book makes a valuable contribution to current literature, and provides a good starting point for a study of structured design. It won't be much help to the application practitioner looking for guidance in selecting and introducing a methodology in his own shop but will sustain the current debate. It is a report not so much on the state of the art, but more on the state of mind of the artists.

K. BOARDMAN (London)