UNIVERSITY OF WARWICK

COMPUTER SCIENCE DEPT

17

MODES IN ALGOL Y

BY

D. J. LEHMANN

Department of Computer Science
University of Warwick
COVENTRY CV4 7AL
ENGLAND

February 1977.

Modes in ALGOL Y

Daniel J. Lehmann
Department of Computer Science
University of Warwick
Coventry, West Midlands. CV4 7AL

1. Introduction.

It recently appeared to the author that the semantics of recursively defined data-types (from now on the term circularly defined data-types will be used to keep away from the "recursive function theory" connotation of the adjective "recursive") could benefit largely from the bulk of recent work on domain equations initiated by Dana Scott.

This note's purpose is : to suggest to people working on the definition of programming languages an alternative way of thinking about circularly defined data types, and to interest the theoretical community to concrete data types, as opposed to the abstract data types of Liskov and Zilles (74), and ADJ (75). The way chosen for that purpose is a reflection on mode definition in ALGOL 68. This language has been chosen because it is the most comprehensive attempt to define a useful, powerful language with extensive type checking; the critical remarks contained in this note are only provoked by the fact that the ALGOL 68 report is the only serious formal attempt to describe mode definition and as such the only one worthy of criticism.

Let the reader be warned that the ALGOL 68 jargon will be used but not exclusively and that the author's critical reading of Tanenbaum's (76) tutorial paper hides only appreciation and admiration.

2. FORTRAN, ALGOL 60 and the untyped procedures.

The concept of a sub-program with parameters as expressed in the FORTRAN subroutines, the ALGOL 60 procedures or the assembler macros has been recognized since the early times of computing to be of cardinal importance. It is the key to clear and pleasant programming (the moderns would say structured). It is certainly with us to stay.

The FORTRAN designers also realized that it was vital that subroutines could accept other subroutines as parameters, thus opening the way to self-application, even in the absence of self-invocation, as in the following example.

```
      SUBROUTINE  F(P)
      IF(.FALSE.)  CALL P
      PRINT  1
      RETURN
    1 FORMAT ('HELLO')
      END
      PROGRAM SELF
      CALL  F(F)
      STOP
      END
```

Another key idea which is with us to stay was introduced in FORTRAN : typing. Each identifier has a type and can only hold "values" of that specified type. All subroutine identifiers have type "subroutine". All these ideas were borrowed by ALGOL 60 in which one new key idea was introduced : circular definition of procedures. The block structure is really only a secondary benefit of circularity. As a consequence self-application could be put to non-trivial use and no simple mathematical object could be seen to be an acceptable semantic domain for procedures.

Most programmers, and certainly the ALGOL 68 group, thought that this was an unexpected result of a careless definition of types in ALGOL 60, was computationally meaningless and should be banned in ALGOL X, the next version of the language.

3. Typed procedures in ALGOL 68.

The principles of orthogonal design and static mode checking of ALGOL 68 implied that procedures should be typed. In the type of a procedure are fixed the number of its arguments, the type of each argument and the type of its value. As a consequence self-application seems to have disappeared. Now the principle of orthogonal design and that of extensibility, spelled out by Tanenbaum (76) implied user-defined modes. Quoting from Tanenbaum :"Another principle, related to that of orthogonality, is the principle of extensibility. ALGOL 68 provides a small number of primitive data types, or modes, as well as mechanisms for the user to extend these in a systematic way. For example, the programmer may create his own data types and his own operators to manipulate them", or later from the same author : "One of the most powerful features of ALGOL 68 is its rich collection of data types (modes), and the facilities it provides programmers to define their own modes".

The application of the principle of orthogonal design now demanded that circular definitions of modes be accepted without any restrictions, as circular definition of procedures is accepted without restrictions. For reasons that we shall try to elucidate later the ALGOL 68 designers departed from their proclaimed policy of orthogonal design and imposed restrictions on recursively defined modes. Before studying this question let us pause and mention two other quirks in the definition of ALGOL 68.

4. Void ?

The revised ALGOL 68 report 0.2.2.(b) claims : "Moreover, in ALGOL 68, a mode-declaration permits the construction of a new mode from already existing ones." We shall come back later to this same quotation because it seems to exclude circularly defined modes but what the author wants to stress now is that it seems to imply that modes are always built from modes by the use of mode-constructors. This is not the case : void is a notion which cannot be derived from the metanotion MODE, nevertheless modes can be built from void, for example : proc (int) void is a mode.

This is obviously a pain to Tanenbaum who explains : "A procedure that is not used as a function, that is, does not return any explicit value, is said to return void." What is probably meant is that such a procedure is said to return an object of "mode" void, though void is not exactly a mode. It would be conceptually much simpler to allow void to be a basic mode, consisting of only one object, from which other modes can be defined. An object of mode proc (int) void would then naturally return as its value, not void, but the unique object of mode void.

5. Nil.

Tanenbaum writes : "In list processing applications, it is necessary to have some marker to indicate the end of a list. When programming in Assembly Language, zero is often used. In ALGOL 68 a special symbol, nil (RR 5.2.4), is provided to end lists."

It is indeed surprising that a concept of ALGOL 68 should be best explained by analogy to machine language. It will be shown in the sequel that proper use of empty lists should enable the user to by-pass this strange nil.

6. Circularly defined modes.

As was indicated at the beginning of Section 4 the ALGOL 68 revised report seems in one place to exclude circularly defined modes altogether.

At another point (RR 4.2) : "Mode-declarations provide the defining-mode-indications , which act as abbreviations for declarers constructed from the more primitive ones or from other declarers or *even* from themselves".

There the report explicitly allows circularity (the author knows that where you can see it is allowed is not in the comments , but in the formal definition, but the formal definition is so obscured by all its restrictions that nobody, except its authors and the implementers, will probably ever bother to read it in full). Yet the "even" seems to indicate mild contempt or more probably instinctive fear from that facility.

Nevertheless both the Report and Tanenbaum acknowledge that the mode-definition facility is essential only when modes are circularly defined.

Later (RR 4.2.*1*.) we find : "The use of 'TALLY' excludes circular chains of mode-definitions such as mode a=b, b=a". This comment is poor in informative value but it means that some circular definitions are forbidden; to know which ones one may work out what the effect of the metanotion TALLY is in the vW-grammar, rely on the taoist explanation of (RR 7.4.*1*.) or trust Tanenbaum who explains : "As you might expect, ALGOL 68 allows all the modes that are intuitively reasonable and prohibits those that are not." Unfortunately the author can only strongly disagree with this last explanation. Here are some examples.

mode fun = proc (fun) fun   is legal. Is it intuitively reasonable?

mode notfun = proc notfun   is illegal. Is it intuitively less reasonable than the previous example?

mode a = a   is illegal. What is wrong with it?

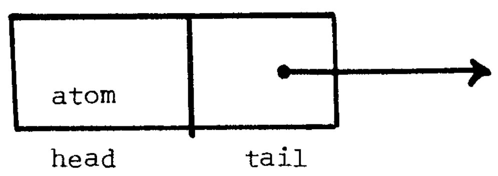
Surely a is not a very useful mode but :

proc f = (int k) int : f (k)   is legal and exactly as useless.

Later it will be shown that all the above examples make perfect sense as defining initial fixpoints of domain equations, but before that let us show how cumbersome and inelegant is, in ALGOL 68, the definition of a mode as simple as a linear list of atoms. Suppose that the mode atom has been defined. The ALGOL 68 way of defining a mode list of atoms would be :

mode listofatoms = struct (atom top, ref listofatoms tail)

The problem is that the mode listofatoms which can be pictorially depicted by :



does not match at all the intuitive idea of a list.

An object of type listofatoms is not a list of atoms it is a possible element in a list of atoms, more precisely a possible implementation of such an element. No wonder that writing programs for list processing will be a very unnatural task.

Moreover, there is no way the empty list of atoms can be made an object of mode listofatoms. As a consequence initializations and termination tests will look weird.

An attractive alternative will be informally presented now, by using arbitrary circular mode-definitions, and supposing void is a mode of which there exist a unique object : constant of type void.

The following definition is totally heretic from the ALGOL 68 point of view but nevertheless, gives a good idea of what a list of atoms should be :

```
mode    latom   =   union  ( void,  struct (atom  top, latom tail) )
```

Notice that the ref has disappeared and that the above definition is a straightforward translation of the idea that a list is either empty or consists of a first atom followed by a list. Now list processing can be performed naturally.

Let us rewrite in this style Tanenbaum's example (3.10).

```
begin
      mode    person  =  struct (string  name, int  score) ;
      mode   lperson  =  union (void, struct (person top, lperson tail) );
      lperson gameresults :=  constant of type void, unlooked;
      bool empty, still looking; string bowler; int bowled ;
      make term (stand in, "  " ) ;
      while read ((bowled, bowler)) ; bowled  > o
      do    gameresults := ((bowler, bowled), gameresults)
      od;
```

```
¢ phase 2.  look up the scores  ¢
while read ((newline, bowler)); bowler ≠ " "
do    still looking := true; unlooked := gameresults; empty := false,
      while still looking ∧ ¬empty
            do case unlooked in  (void)  : empty := true,
                                 (struct (person top,lperson tail)list):
            if name of top of list = bowler
               then print ((bowler, score of top of list, newline));
               else unlooked := tail of list
            fi
               esac
            od
            if still looking
               then print ((bowler, "not in our league", new line))
            fi
od
end
```

The reader should notice that by introducing a unique constant of type void one has acheived the objective of having only one empty list (an empty list of atoms is also an empty list of books), without having to play with a nil object of undefined mode.

The difference in. mode between the empty list and non-empty lists reflects the fact that the operations available on non-empty lists (top and tail) are not available on the empty list.

A more general notion of a list (like the one used in LISP) could be defined by :

<u>mode</u> <u>glatom</u> = <u>union</u> (<u>void</u>, <u>struct</u> (<u>union</u> (<u>atom</u>, <u>glatom</u>) head,
<u>glatom</u> tail))

which defines as clearly as possible a generalized non-empty list to be composed of a head which is either an atom or a list, and a tail which is a list. This definition looks very different than the one which would be suggested by ALGOL 68 :

<u>mode</u> <u>alglist</u> = <u>struct</u> (<u>alglist</u> head, <u>alglist</u> tail)

which in fact defines unlabelled binary trees. It is clear that there is some kind of equivalence between generalized lists and binary trees; nevertheless it seems very unwise to force the programmer to use binary trees when he is thinking in terms of lists. The binary tree representation of a one element list is, for example, very unnatural.

7. References, pointers and list processing.

One of the main innovations of ALGOL 68 is the introduction of modes of type <u>ref</u> "something". Is this novelty an asset or a liability? Tanenbaum first justifies extensively the introduction of reference-to modes by stressing the difference between constants and variables and insisting that certain formal parameters should be specified as variables (those which are assigned to) and that others should be specified as constants. But later he goes on : "Variables involving <u>ref</u> "something" are typically used in list processing applications". But list processing is not specially concerned with the distinction between assignable and non-assignable formal parameters. What is then the real reason for introducing modes of type <u>ref</u> "something" ? The distinction between constants and variables could have been brought up to light in a much less general way and it seems to the author that the introduction of references in ALGOL 68 is mainly intended for list processing. In fact list processing would be impossible in ALGOL 68 without explicit pointers and variables of mode ref ref "something". It seems indeed that the ALGOL 68 designers thought that pointers had to be used in list processing applications. One should, on the contrary, reflect on the fact that LISP, widely used for list processing, does not have pointers at all. Indeed one of the reasons of the success of LISP as a programming language is that the user does not have to manipulate pointers. To a similar effect Milne and Strachey write : "Cyclic data structures which are declared by incidence rather than by reference may be useful as they have a greater degree of inviolability than those containing locations which may be subject to assignments."

Pointers are probably necessary when one wants to write extremely efficient programs in a machine language-like manner but the user should be given the opportunity to use data-structures without having to introduce the possibility of shared values which is bound to complicate enormously the validation of a program. At the risk of repeating himself, the author wants to stress that if LISP is a programming language which is so widely used for writing large complex and correct programs (especially in artificial intelligence) it is due both to its high functionality which ALGOL 68 retains, and to the security given by the absence of shared values, which is unattainable in ALGOL 68. It is the author's belief that the introduction of references should not be seen as obliviating the need for circularly defined modes of the type described above.

8. <u>Arbitrary circular mode definitions are meaningful.</u>

It is left to show that arbitrary circular mode definitions are meaningful. That this is so follows from the pioneering work of Scott (71, 72 and 76), explicited in Wand (74) and Lehmann (76). This is not the place to expose the mathematics of this approach but it can be said that, in complete analogy with circular definitions of functions, a circular mode definition can be seen as defining an initial solution to an equation of type X=T(X) over a suitable category of domains, and for a suitable functor T. The initial solution comes in the form of an object in the category of domains D and two inverse isomorphisms between D and T(D) : $D \underset{\psi}{\overset{\phi}{\rightleftarrows}} T(D)$ satisfying a certain universal property. The isomorphisms ϕ and ψ equip D with operations and make D an (universal) algebra.

A very general category of domains Dom is defined in Lehmann (76) and a large number of sub-categories of Dom can be seen to be adequate too. The basic data-types : integers, reals, booleans can easily be considered as objects in the category of domains. One more basic data-type should also be considered : 1 consisting of only one object (it is the initial object in the category of domains and corresponds to the ALGOL 68 <u>void</u>). The main mode constructors can be interpreted as functors in the category of domains : <u>proc</u> is $\rightarrow$, <u>struct</u> is x, <u>union</u> is $\oplus$, as is shown in Lehmann (76). A mode definition (circular or otherwise) is then interpreted as defining the initial fixpoint of a domain equation. For the interpretation of the isomorphisms see example 4 below.

The *P* functor of Lehmann (76) suggests a powerset mode constructor which does not exist in ALGOL 68. The mode constructor <u>row</u> could be expressed as a <u>proc</u> ; <u>ref</u> can only be given a mathematical meaning in the presence of a model of the store.

9. Examples

1) mode fun = proc (fun) fun

   defines the initial fixpoint of $D \cong [D \to D]$ which is 1, the one point domain.

2) mode notfun = proc notfun

   defines the initial fixpoint of $D \cong [D \to 1]$ which is 1 also.

3) mode a = a

   defines the initial fixpoint at $D \cong D$ which is 1.

4) mode latom = union (void, struct (atom head, latom tail))

   defines the initial fixpoint of $D \cong 1 \oplus ATOM \times D$ and gives names to part of the isomorphisms.

   Let $S \underset{\psi}{\overset{\phi}{\rightleftarrows}} 1 \oplus ATOM \times S$ be the initial fixpoint of the above equation. Clearly ϕ sends the empty list to the unique object of 1 and the non-empty lists into a pair consisting of their top and their tail. Its inverse ψ sends the unique of 1 to the empty list and every pair of an atom and a list into the list resulting from pushing the atom on the list.

   Symbolically $\phi(p)$ = if empty (p) then $\perp_1$ else top(p) x tail(p)

   ψ = null $\oplus$ push

Note 1: In examples 1) and 2) the procedures have been considered to be without side-effects and global variables, which was a gross simplification. To be more general would involve introducing Environments and Stores as in Milner-Strachey.

Note 2: Arbitrary circular definitions are now available in any data-types, not only those of type procedures.

10. Implementation.

To adopt a policy of unlimited use of circular mode-definitions would involve a departure from the half-stated policy of ALGOL 68 which is that mode-definitions define templates for storage allocation and of checking for equivalent modes.

On the first point one can only notice that mode-definitions do not anyway define templates known to the user. As Tanenbaum writes : "When an integer variable acquires a new value,as in *i*:=3 , the bit pattern for the integer 3 is put into location *i*. Obviously, assigning sin to f (f:=sin) is not going to cause a copy of the procedure's machine code to be stuffed into the variable f. The ALGOL 68 compiler writer must determine how to implement this, but presumably he will assign pointers to the procedure's code and environment (or the equivalent) to f." And later he says :"You may be wondering how unions are implemented. Presumably the compiler will have to reserve enough space in a united variable for the largest of the alternatives (or if that is too painful, perhaps only a pointer will be stored)."

Clearly the user, when he defines new modes, does not have to know how they will be stored and sometimes hidden pointers are involved. This is very much to the taste of the author who tends to think that all pointers should be hidden from the user.

On the second point the author agrees with those many people who think that mode-equivalencing should not take place at all. Non-equivalencing of different modes allows the user to use the type-checking facilities in order to ensure correctness of his programs.

Once it is admitted that mode definitions do not directly yield templates for storage and that there should be no equivalencing of modes a programming language with full capabilities for circular definitions can be implemented.

11. Conclusion

The above remarks were aimed at showing that a programming language as powerful as ALGOL 68 and with more general mode-definition facilities can be defined with a very natural semantics.

BIBLIOGRAPHY

ADJ : Goguen, Thatcher, Wagner and Wright (1975) "Abstract data types as initial algebras and the correctness of data representations" Proceedings, Conference on Computer Graphics, Pattern Recognition and Data Structures, May 1975.

Lehmann, Daniel J. (1976) "Categories for fixpoint semantics". Theory of Computation Report No.15 University of Warwick, Dept. of Computer Science.

Liskov-Zilles (1974) "Programming with abstract data bypes". SIGPLAN Notices 9.

Milne-Strachey (1976) A theory of programming languages semantics. Chapman and Hall.

Scott, Dana S. (1971) "The lattice of flow diagrams". Semantics of Algorithmic Languages (E. Engeler, ed.) Springer Lecture Notes in Mathematics, vol.188 (1971) pp. 311-368.

Scott, Dana S. (1972) "Continuous lattices". Toposes, Algebraic Geometry and Logic. (ed. by F.W. Lawvere). Springer Lecture Notes in Mathematics, vol.274 (1972), pp.97-136.

Scott, Dana S. (1976). "Data types as lattices". SIAM Journal on Computing. vol.5.

Tanenbaum, Andrew s. (1976) "A tutorial on ALGOL 68" ACM Computing surveys. Volume 8. June 1976.

van Wijngaarden, A; Mailloux, B.J.; Peck J.E.L.; Koster, C.H.A.; Sintzoff, M.; Lindsey, C.H.; Meertens, L.G.L.T.; and Fisker, R.G. (1975). "Revised report on the Algorithmic Language Algol 68", Acta Informatica 5,(1975), 1-236.