ma
the
ma
tisch

cen
trum

AFDELING INFORMATICA IW 45/75 DECEMBER

L.G.L.T. MEERTENS

MODE AND MEANING

Prepublication

amsterdam 1975

stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 45/75 DECEMBER

L.G.L.T. MEERTENS

MODE AND MEANING

Prepublication

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

AMS(MOS) subject classification scheme (1970): 68A05

ACM-Computing Reviews-categories: 4.22, 5.24

Mode and meaning [*)]

By

L.G.L.T. Meertens

ABSTRACT

The concept of the (static) *mode* of a value is considered in relation to the (dynamic) *meaning*. It is shown how this concept may be carried further in order to increase the security and to obtain higher levels of abstraction.

KEY WORDS & PHRASES: *Mode, meaning, security, levels of abstraction.*

---

*) This paper is not for review; it is meant to be published elsewhere.

0. INTRODUCTION

Values are usually represented inside a computer by a sequence of bits. The meaning of such a sequence cannot be derived from the sequence alone. For example, an integer and a pointer might happen to be represented by identical strings of 0's and 1's, even though their meanings, as far as the programmer is concerned, are vastly different. One of the major innovations marking the advent of high-level programming languages is the concept of a *mode* (or *type*) associated with values. This constituted a recognition of the preponderance of certain well understood meanings assigned by programmers to bit sequences. Thus, one may have real numbers, of mode "real", or truth values, of mode "boolean", and the syntax prevents unintentional confusion between values of these two modes. Moreover, the representation is hidden from the programmer; this is in itself quite useful in order to achieve abstraction from the particular representation employed, with all the benefits inherent in such abstraction.

This concept of mode seems to have reached its climax, at least up until now, in ALGOL 68. For this reason, this paper uses ALGOL 68 as a point of departure. Its purpose is to indicate how the concept may be carried even further in order to increase the security by more comprehensive static checking, and even to obtain higher levels of abstraction. The presentation is certainly not a coherent, theoretical one, but, rather, informal and at times even haphazard. New ad-hoc syntactic constructions may at times be introduced without warning, and with semantics only suggested by context. I hope it will be clear that these are not put forward as proposed extensions of ALGOL 68. Should the ideas outlined here provoke some reflexion from language designers, this paper will have served its purpose.

1. THE GREATER SECURITY OF ALGOL 68 - AN EXAMPLE

The ALGOL-68 concept of the mode of a value is a full-grown version of the type of ALGOL 60. The Revised ALGOL 68 Report [1] states (in pragmatics): *Each value has an attribute, termed its "mode", which defines how that value relates to other values and which actions may be applied to it.* Although the ALGOL-60 type gives a hint in the same direction, it does not convey sufficient information to determine statically whether a given operation (i.e., action applied to a value) will be meaningful. As an example, consider the following ALGOL-60 program:

```
l: begin integer i;
         procedure var (op); procedure op; op (i);
         procedure spot (op); procedure op; op (l);
         procedure zero (v); integer v, v:= 0;
         procedure jump (s); label s; go to s;
         procedure do (op, arg); procedure op, arg; arg (op);
         do (zero, var);
         do (jump, spot)
   end.
```

The execution of the two procedure statements that call *do* is perfectly meaningful. With the same ease, however, it is possible to write *do (jump, var)*, *do (zero, spot)* or even *do (do, do)*, whose execution would result, respectively, in the meaningless operations <u>*go to*</u> *i*, *l:= 0* and *do (do)*. Note that this possibility of ALGOL 60 is not due to specification being optional, but to the undiscriminating specification <u>*procedure*</u>, which does not disclose the type of parameters accepted (LANGMAACK [2]).

In ALGOL 68, *zero* and *jump* would have different modes, since they accept parameters of different modes. Similarly, since the *op* parameters of *var* and *spot* have to accept different modes for their parameters, they are themselves of different modes and so then are *var* and *spot*. Consequently, the *arg* parameter of *do* would have to be "specified" as accepting two different modes, i.e., the "union" of these modes. The same holds, of course, for the *op* parameter. Before *arg* may be called with *op*, the different cases have to be distinguished in order to see if the actual modes comply. A comparable ALGOL-68 program would therefore run as follows:

```
l: begin int i;
        proc var = (proc (ref int) void op) void: op (i);
        proc spot = (proc (proc void) void op) void: op (go to l);
        proc zero = (ref int v) void: v:= 0;
        proc jump = (proc void s) void: s;
        proc do = (union (proc (ref int) void,
                          proc (proc void) void) op,
                   union (proc (proc (ref int) void) void,
                          proc (proc (proc void) void) void) arg)
        void:
        case arg in
         (proc (proc (ref int) void) void pprivv):
              case op in
               (proc (ref int) void priv): pprivv (priv)
              out error 1
              esac,
         (proc (proc (proc void) void) void pppvvv):
              case op in
               (proc (proc void) void ppvv): pppvvv (ppvv)
              out error 2
              esac
        esac;
        do (zero, var);
        do (jump, spot)
  end.
```

The definition of *do* is rather complicated compared to the ALGOL-60 version. What has happened is that the type-checking implied in a foolproof ALGOL-60 implementation has been made explicit here. Such calls as *do (jump, var)* and *do (zero, spot)*, when elaborated, give rise to the elaboration of *error 1* and *error 2* (not further defined here), respectively. An attempted call like *do (do,* *do)* is even syntactically invalid. Thus, there is no way of writing a (syntactically valid) call with a parameter whose value is of a potentially unacceptable mode, except for the case where an explicit test is made before that value is used. (It is, perhaps, also illustrative that in a first approach I inadvertently wrote *op (arg)* for the body of *do* and only noticed the error while transcribing the ALGOL-60 version to ALGOL 68.) Also, implicit tests are no longer needed. So, as far as both safety and efficiency are concerned, the ALGOL-68 approach is superior to that of ALGOL 60.

2. MODE AS A STATIC SUMMARY OF MEANINGFUL OPERATIONS

The mode of a value may be viewed as a static summary of the meaningful operations on that value. For example, if a list consists of two fields, the *head* field, which is a string, and the *tail* field, which is a pointer to a list, the following mode-declaration reflects this recursive definition:

mode list = struct (string head, ref list tail).

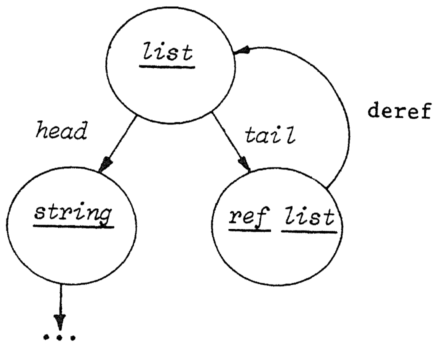
This declaration may then also be taken as stating that meaningful operations on a list are:

- selection of the *head* field, followed by meaningful operations on a string;
- selection of the *tail* field, followed by meaningful operations on a pointer to a list.

If we take as meaningful operations on a pointer to a list:

- following the pointer (dereferencing), followed by meaningful operations on a list,

then this verbal explanation may be depicted schematically in the following diagram:

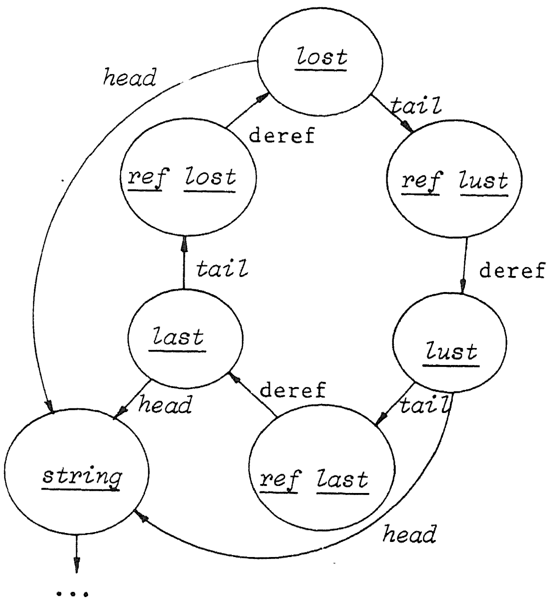


This diagram corresponds to a finite state automaton which, with *list* as initial state, accepts a (regular) language, each of whose sentences corresponds to a meaningful operation on a list.

For the more complicated mode-declaration

mode lost = struct (string head, ref lust tail),
lust = struct (string head, ref last tail),
last = struct (string head, ref lost tail),

we obtain the following diagram:



The corresponding finite state automaton, with *lost* as initial state, accepts the same language as the automaton for *list*: the meaningful operations on *list* and *lost* values coincide. Indeed, *list* and *lost* (or *lust* or *last*) specify the same mode, as is consistent with the philosophy of modes in ALGOL 68, quoted above. The connection between the equivalence of modes and of finite state automata has already been pointed out by KRÁL [3].

3. HOW TO FILL THE REMAINING GAPS

Until now it has been taken more or less for granted that the operations prescribed by a syntactically valid construct are always (dynamically) meaningful. This need not, however, be the case. I shall consider two cases in which "the further elaboration is undefined":

- A name which is to be dereferenced, or to which another value is to be assigned, happens to be nil (as in *<u>ref</u> <u>real</u> (nil):= 3.14*).

- The subscript in a slice is out of range (as in *[1 : 3] <u>int</u> a; a[4]*).

It is a desirable property of programming languages that such cases cannot arise, for more than one reason: If we succeed in excluding all such cases, the possibility of writing (syntactically valid) programs whose elaboration turns out to be undefined only at run-time simply disappears. If the programmer cannot show statically that a certain operation will be meaningful, the burden of testing is laid upon his shoulders. This will, in general, increase the number of marks he has to write down (cf. the definition of *do* in ALGOL 68). The programmer is thereby encouraged to reconsider the problem, and to reformulate his implementation of the algorithm in such a way that it becomes clear that undefined operations cannot arise. The understandability of the program will then, hopefully, increase. Moreover, he is obliged to consider each case in which things might go wrong. A further, not insignificant, advantage is the disappearance of costly implicit tests in a foolproof implementation.

Although only two special cases are discussed below, most of the other existing gaps may be approached in much the same spirit. For example, a syntactic check on the use of uninitialized variables is present in ALEPH [4,5]. Even the problem of wrong input data may perhaps be solved by specifying, as part of the program, a syntactic description against which the input is checked beforehand.

3.1. *The problem of nil names*

A complete and satisfactory solution of the case of dereferencing or assigning to a nil name is possible within the framework of ALGOL 68. It suffices to exclude nil names (e.g., by not providing a representation for the nil-symbol), so that any name refers to a value. If a programmer wants to use constructs whose value may be a name, but may also not refer to a value (i.e., be nil), he can write a construction like

```
mode list = struct (string head, union (ref list, void) tail).
```

A nil list is then represented by the value empty, the only void value. A piece of program such as

```
ref list a:= z;
while tail of a :≠: ref list (nil)
do a:= tail of a od
```

would have to be rewritten

```
ref list a:= z;
while case tail of a in
```

```
            (ref list b): (a:= b; true)
         out false
         esac
      do skip od.
```

Note that the number of tests explicitly performed remains the same, and that the total number of tests decreases, since in a not exceedingly intelligent implementation the assignation *a:= tail of a* entails a test on *ref list (a)* not being nil (because of the selection), and perhaps even two more tests on *a* itself (because of the assignment and the implied dereferencing of the second occurrence).

The rewritten version admittedly looks a bit cumbersome. It is my feeling that future programming languages should have a more convenient way of expressing such a flow of control. A possible notation might be

```
      ref list a:= z;
      do tail of a in (ref list b): a:= b od.
```

There is also a connection with DIJKSTRA's guarded commands [6].

3.2. *The problem of the subscript range*

The treatment of subscripts is not dealt with so easily. Yet this case seems typical both of the inefficiency caused by high-level programming languages compared to hand coding - in most cases the programmer knows perfectly well that the subscript is in range, but he is unable to exploit this knowledge - and of the opportunities provided by even high-level languages for writing programs whose lack of meaning transpires only during elaboration - it does not seem bold to conjecture that "subscript out of range" accounts for about half of the run-time errors. A first step towards a solution has been made in PASCAL, by introducing ranges as a type. The major defect of this approach is that ranges are completely compile-time determined, just as in good old FORTRAN. For a more detailed criticism, see HABERMANN [7].

The obvious next step requires modes which can be yielded dynamically, but which are staticized over certain parts of the program. Suppose we may write:

```
      mode r3 = 1 : 3;
      [r3] real a;
      for i ∈ r3 do a[i]:= 0 od.
```

Used in this way, this is a mere abbreviation, but it should also be possible to write directly

```
mode r = 1 : n; [r] real p; read (p);
for i over r
do r imin:= i; real min:= p[imin];
   mode r2 = succ i : upb r;
   for j over r2
   do if p[j] < min then min:= p[imin:= j] fi od;
   p[imin]:= p[i]; p[i]:= min
od.
```

This requires that the syntax recognizes in some way that the range yielded by *succ i : upb r* is a subrange of *r*, provided that *i* is of mode *r*. This may seem a bit ad hoc, but it is, in principle, certainly possible. In fact, what should be recognized by the syntax and what should not can only be determined by studying a number of representative, diverse algorithms. It is probably too much to ask for recognition of the fact that $(i + j) \div 2$ lies in the same range as i and j. In such cases, we always have the escape afforded by

```
case (i + j) ÷ 2 in (r mid): p[mid] out skip esac.
```

This is acceptable as long as the number of cases where such cumbersome expressions cannot be circumvented is sufficiently small.

Some statistics were obtained by examining 24 small programs, taken from various courses on ALGOL 68. Out of these 24, 16 programs used multiple values (where the use of strings as though they were primitive values was not counted). A total of 95 slices were found, 88 of which could be expressed with static range checking without the need to introduce an extra dynamic check. In this count, not only were simple loops over a range included, but also cases like

```
while t ≤ n do t:= a[t] od,
```

which can be expressed, using the notation suggested at the end of section 3.1, as

```
do t in (r t1): t:= a[t1] od.
```

On closer inspection, 2 of the 7 remaining slices (both occurring in a textbook in print!) might give rise to a subscript-out-of-range error. So, if we only count the other 5 cases as genuine, the quite acceptable percentage of 5% is obtained for the cases where slicing is encumbered by the proposed measures.

Some features which might further reduce this percentage:

- For-loop in reverse order (*for i revo r do ... od*).
- Static recognition that certain parts of the program are reachable only if

some range *r* is not empty, so that both *lwb r* and *upb r* are known to be in range, as at the "..." in *for i over r do ... od*.

- Allowing intersection of ranges, as in *mode r* = *r1* ∩ *r2*, after which *r* is recognized as a subrange of *r1* and *r2*.

4. MODES DEFINED BY CHARACTERISTIC PREDICATES

BAUER & GNATZ [8] have suggested the possibility of definitions like

mode posreal = *(real x) bool*: $x > 0$.

This is quite similar to *mode m* = $1 : n$, which could be expressed by

mode m = *(int i) bool*: $1 \le i \wedge i \le n$.

Of course, in the context of *real x*, *posreal y*, *z*, the assignations *x*:= *y* and *y*:= *z* are OK, but *y*:= *x* is not, since it cannot be guaranteed that *x* is positive. One has to write

case x in (posreal p): y:= p out error esac.

The reason for mentioning this construction here is that it provides a tool for increasing security in certain cases where no other simple syntactic means avail. Suppose that a function has been defined which only converges if a positive argument is supplied. A sensible programmer may insert a check at the routine's entry, but if this check fails, his trouble begins. Where did the erroneous value originate? How was it passed on? By prescribing the mode *posreal* at all places where he knows the values involved should be positive, the programmer can catch the error at its point of origin.

5. ABSTRACTION FROM DATA REPRESENTATION

Even though a certain operation may be perfectly meaningful as far as the semantics of the algorithmic language is concerned, it may be sheer nonsense from a human point of view. My school teachers would insist that one cannot add apples and pears together, and yet enough programming errors have been made by adding birth dates to salaries or relative errors to absolute quantities. I myself have a sore recollection of a program dealing with music which employed two different systems to represent notes by integers. In spite of some careful measures, values in one system kept turning up at places where values in the other system were expected.

It is strange that the security provided by abstraction from data representation is offered at a low level, but should be withheld from the user at higher levels of abstraction.

An example of the use of integers to represent an in fact completely different concept is given by

```
int male = 1, female = 2, neuter = 3.
```

The intention here is merely to introduce three new, distinct values to represent genders. This is much more clearly expressed by the scalar type of PASCAL:

```
type gender = (male, female, neuter).
```

In this way, meaningless expressions like *male + female* × *neuter* are prohibited. (The formula *6 - me - de* in the Towers-of-Hanoi example of the Revised ALGOL 68 Report is a "dirty" programming trick to select the third element of a set, different from *me* and *de*.)

An elegant way to prevent the mixing up of arithmetic quantities representing, e.g., physical quantities with different dimensions, is presented by CLEAVELAND [9]. His system is described by grafting it on ALGOL 68, using the same descriptional technique as the Revised Report. A simple example is given by

```
pouch inch real x, y, pouch cm real z.
```

The declarer *pouch inch real* is used for values which are represented as real numbers but have a distinct mode, as indicated by the tag *inch* which, of course, has no inherent meaning. After these declarations, *x:= y* is a valid assignation, but *x:= z* is not. It is possible, however, to write *x:= z* × *0.3937 inch/cm*. The validity is checked by performing dimensional analysis as part of the mode mechanism. (Note that *0.3937 inch/cm* is not a formula, but a denoter for values of the mode *pouch inch/cm real*.) A quite different method of use is illustrated by

```
[1952 : 1975 years] bool of age;
for y from 1952 to 1975 years
do of age [y]:= y - 1952 years ≥ 21 years od.
```

For more-dimensional multiple values, association of different pouches with different subscript positions will effectively prevent accidental transposition of subscripts.

Although the incorporation of such features in future programming languages should be encouraged, the general problem needs a separate attack. There will always remain a need for the introduction of new modes with a meaning not yet catered for. In order to represent values of that mode, some choice has to be made among the already definable modes. Presumably, some operations are then possible which are meaningless in terms of the intended meaning.

Consider, e.g., a mode "stack". Meaningful operations on a stack are pushing and popping. In order to model a stack, a pair consisting of an integer and a large array might be used. Alternative representations are a linear list or a double-ended queue (which may be available as primitive in some programming languages). Using the last representation it would be possible to pop the bottom of the stack, an operation which cannot be expressed reasonably using the first model. Conversely, using the first model it is possible to select elements of the stack at random. All of these are meaningless in terms of the semantics of a stack and should be precluded.

Using the class concept of SIMULA 67, we can define

```
class stack;
begin integer ptr; integer array cell[1 : large];
      procedure push (i); integer i;
      begin ptr:= ptr + 1; cell[ptr]:= i end;
      integer procedure pop;
      begin pop:= cell[ptr]; ptr:= ptr - 1 end;
      ptr:= 0
end.
```

If, as has been proposed, the identifiers *ptr* and *cell* are hidden and only *push* and *pop* are exported, then, after the declaration <u>ref</u> *(stack) s, t*, the only operations that may be used on *s* and *t* are *s.push* and *s.pop*, and *t.push* and *t.pop*, respectively. Thus, in fact, each new variable of class *stack* is of a new, unique mode.

The advantages of the use of classes for lightening the correctness proof by factoring it are expounded by HOARE [10]. A very similar construction is put forward by LISKOV & ZILLES [11], who stress the fact that an abstract data type should be characterized by the available operations as defined by the user. The germ of this idea could already be found in WILKES [12].

This feature, nice though it may be, is insufficient to cater for our needs, since only monadic operations may be prescribed. A solution has been proposed by MORRIS [13], which will be demonstrated here as an extension of "definition modules" as proposed by SCHUMAN [14]. There, identifiers and other indicators may be hidden in the context in which the definition module is invoked by preceding their definition by *implicit*. If, likewise, the possibility is introduced of prefixing a mode-definition with *primitive*, meaning that in the invocation context the mode defined is treated as primitive, we achieve the desired effect. For example, we might define

```
definition prelude =
begin mode primitive int = [0 : 15] bool,
           primitive char = [0 : 15] bool,
      op + = (int a, b) int: ...,
      ...,
      op abs = (char a) int: a,
      op repr = (int a) char: a
end
```

and then invoke the definition in

```
begin env prelude;
      int i, char c;
      ...
end,
```

where in the latter context *i:=* abs *c* and *c:=* repr *i* are legal assignations, while *i:= c* and *c:= i* are syntactically invalid. The mode of abs is proc ([ ] bool) [ ] bool in the context of the definition module, but proc (char) int in the invocation context.

A recurrent complaint with respect to the design of ALGOL 68 has been the use of the name concept both for variables and for pointers to (structured) objects. The essence of this complaint is the fact that two properties of names are invariably linked together: that of granting access to an object without automatic creation of a copy, and that of granting the right to supersede it (or its possible sub-objects) with another object. For example, if lists are represented in the usual way, access to a list implies the possibility of an operation such as *tail* of *a:= a*, creating an ill-formed (circular) list. This is a typical example of a meaningless operation in terms of the semantics of lists, made possible by the fact that some representation has to be chosen. Using the above approach, one may define

```
definition lists =
begin mode list = union (nil, properlist),
           primitive nil = void,
           primitive properlist = ref struct (string head, list tail),
      op * = (string a, list b) properlist:
      heap struct (string head, list tail):= (a, b),
      op head = (properlist a) string: head of a,
      op tail = (properlist a) list: tail of a,
end.
```

Only well-formed lists may be formed with the operations made available by this module. In other words, if a language such as ALGOL 68 is endowed with the possibility of user-defined "primitive" modes, the basis for complaints like the above vanishes completely.

6. CONCLUSION

A sketch has been given of how information about dynamic properties of values (their meaning) can be expressed as a static property (their mode), with varying degrees of sophistication. It is my belief that it is not sufficient to study disciplines through which more reliable programs may be obtained, but that we should also provide tools through which the willing spirit can enforce this discipline on his weak flesh. I hope that this paper may suggest some feasible tools to future language designers.

REFERENCES

[1] WIJNGAARDEN, A. VAN, et al. (Eds.), *Revised report on the algorithmic language ALGOL 68*, Acta Informatica 5 (1975) 1-236.

[2] LANGMAACK, H., *On correct procedure parameter transmission in higher programming languages*, Acta Informatica 2 (1973) 110-142.

[3] KRÁL, J., *The equivalence of modes and the equivalence of finite automata*, ALGOL Bulletin 35.4.5 (1973) 34-35.

[4] BOSCH, R., D. GRUNE & L. MEERTENS, *ALEPH Manual*, Report IW 17/74 (Mathematical Centre, Amsterdam, 1974).

[5] BOSCH, R., D. GRUNE & L. MEERTENS, *ALEPH, A Language Encouraging Program Hierarchy*, The International Computing Symposium 1973, Davos (North-Holland, Amsterdam, 1974) 93-99. (Report IW 9/73, Mathematical Centre, Amsterdam, 1973.)

[6] DIJKSTRA, E.W., *Guarded commands, nondeterminacy and formal derivation of programs*, Comm. ACM 18 (1975) 453-457.

[7] HABERMANN, A.N., *Critical comments on the programming language PASCAL*, Acta Informatica 3 (1973) 47-57.

[8] BAUER, F.L. & R. GNATZ, *Mengen in algorithmischen Sprachen oder: Arten und Prädikate*, Bericht Nr. 7202, Abt. Mathematik der TU München (1973).

[9] CLEAVELAND, J.C., *Pouches, a programming language construct encouraging redundancy*, UCLA-ENG-7555 (1975).

[10] HOARE, C.A.R., *Proof of correctness of data representations*, Acta Informatica 1 (1972) 271-281.

[11] LISKOV, B. & S. ZILLES, *Programming with abstract data types*, Proceedings of a Symposium on Very High Level Languages, SIGPLAN Notices 9 (April 1974) 50-59.

[12] WILKES, M.V., *The outer and inner syntax of a programming language*, Computer J. 11 (1968) 260-263.

[13] MORRIS, J.H., *Types are not sets*, SIGPLAN Symposium on Principles of Programming Languages (1973) 120-124.

[14] SCHUMAN, S.A., *Toward modular programming in high-level languages*, ALGOL Bulletin 37.4.1 (1974) 30-53.