STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

MR 114

On infinite modes.

(Algol Bulletin, (1969), Nr 30, p 86-89.)

by

C.H.A. Koster

1969

AB30.3.3 On infinite modes

C.H.A. Koster
Amsterdam, febr 1969

0. Modes

In ALGOL 68 the 3 types of ALGOL 60 have been replaced by an unlimited number of modes. A mode is any terminal production of the metanotion MODE. The metaproduction rules for MODE form a context free grammar which has an infinite number of terminal productions. In an ALGOL 68 program the number of modes that are actually used is always finite, since all modes that are used have to be the mode specified by some declarer of the program, or 'reference to' such a mode, or the mode enveloped by the original of some denotation, and the number of such declarers and denotations is, of course, finite. The programmer has the freedom to declare new modes by means of a mode-declaration; e.g., he can use lint as a shortening for long int by

E1) mode lint = long int

or manipulate dates as a whole using

E2) mode date = struct (int year, month, day)

Here the modes specified by lint and by date are clearly finite terminal productions of MODE and thus of MOID. A production rule (R6.1.1.2)

E3) SORTETY MOID unit: SORTETY unitary MOID clause.

implies, amongst others,

E4) SORTETY integral unit: SORTETY unitary integral clause.
E5) SORTETY long integral unit: SORTETY unitary long integral clause.
E6) SORTETY [date] unit: SORTETY unitary [date] clause.

where "[date]" stands for the mode specified by date.
Since it is allowed to declare a mode in terms of itself, it is possible for a declarer to specify an infinite mode, e.g., the declaration

E7) mode person = struct (int a, ref person f)

causes person to specify the mode [person] where "[person]" stands for 'structured with integral field letter a and reference to [person] field letter f'. Also for this infinite mode, E3 implies

E8) SORTETY [person] unit: SORTETY unitary [person] clause.

Forbidding the infinite modes would seriously diminish the power of the language, since it would, e.g., make it impossible to have chains of linked structures like

E9) mode cell = struct (string contents, ref cell next)

where the last cell of a chain will have next equal nil.

1. Potential dangers of infinite modes

In allowing infinite modes one must beware of the following:

a. A value of any mode must not take infinite storage space in the computer. Example:

E10) mode full = struct ([1:10] full f, int s)

This declaration is forbidden by R4.4.4.c, since the actual-declarer struct ([1:10] full f, int s) "shows" full. Every value of mode [full] is composed of ten such values and an integer, in this way taking up infinite room in the computer. An allowed declaration however is

E11) mode full = struct (ref [1:10] full f, int s)

because a reference to a row of full's need not take infinite room in the

computer; every reference will take up only very few memory locations, depending on the implementation. And indeed, according to R4.4.4.c, E11 is not forbidden.

b. No mode may allow ambiguous parsing of the program. Consider the clause

E12) (mode t = ref t; t t1, t2; t1:= t2)

In the assignation, the source t2 can be dereferenced any number of times, and still the assignation would be syntactically correct; so the source is ambiguous. Again, this declaration is not allowed by R4.4.4.c, since the actual-declarer ref t shows t.

c. It must be decidable whether two modes used in the program are the same. Compare:

E13) mode person = struct (int age, ref person father)
E14) mode purson = struct (int age, ref purson father)

and even

E15) mode parson = struct (int age, ref
struct (int age, ref parson father) father)

If one tries writing out the modes specified by person, purson and parson by a process akin to developing (R7.1.2.b), one gets after n steps

E16) [person] = A ↑ n [person] B ↑ n

where A = 'structured with integral field letter a letter g letter e and reference to' and B = 'field letter f letter a letter t letter h letter e letter r', and the exponentiation denotes repeated concatenation,

E17) [purson] = A ↑ n [purson] B ↑ n
E18) [parson] = AA ↑ n [parson] BB ↑ n

After an infinite number of steps one would have

E19) [person] = A ↑ inf B ↑ inf
E20) [purson] = A ↑ inf B ↑ inf
E21) [parson] = AA ↑ inf BB ↑ inf = A ↑ inf B ↑ inf

It is clear that the assumption [person] = [purson] does not lead to a contradiction, nor does [person] = [parson].
The purpose of this note is to give an algorithm to decide the equivalence of the modes specified by two given declarers in a program.

2 Equivalence of modes

In the sequel we assume that in all declarers the virtual-, actual- and formal-row-of-rowers (R7.1.1.r) have been deleted, so that the declarers contain no strict-lower- or upper-bounds, etc.
The "defining declarer" of a mode-indication is the actual-declarer following the equals-symbol following the defining occurrence of that mode-indication (R7.2.1.a), e.g., the defining declarer of lint is long int (E1).
A declarer D is "expanded" by replacing in D a mode-indication by its defining declarer. Expanding a declarer does not change the mode specified by it.
The "full expansion" of a declarer D is the declarer obtained by expanding D until no mode-indications are left in it. The full expansion of a declarer which specifies a finite mode can be obtained finitely, but the full expansion of a declarer specifying an infinite mode is an infinite tree (. See K2.3.5.11).

Definition
Two declarers are "equivalent", i.e., specify one same mode, if their full expansions are the same tree.

E22) mode a = ref b, b = real, c = ref real
The full expansion of the defining declarers of both a and c is ref real, so a and c specify one same mode.
E23) mode d = proc d, e = proc proc e
Whether the full expansions of d and e are the same tree depends, after expanding once, on whether the full expansions of proc d and proc proc e are the same tree, and so whether those of d and proc e are the same tree, and so whether those of proc d and proc e are, and so whether those of d and e are. Thus, by induction, it is found that d and e are equivalent.
The induction steps can be formulated:
Two mode-indications specify one same mode if, under the assumption that they do, their defining declarers specify one same mode.
A mode-indication and a declarer specify one same mode if, under the assumption that they do, the declarer specifies the same mode as the defining declarer of the mode indication.
This approach to the equivalence of declarers suggests the following algorithm for deciding the equivalence of two declarers:

3. Algorithm

Step 1 If the two given declarers D and E are the same sequence of symbols, then they are equivalent;
Step 2 if D is a mode-indication, then D and E are equivalent provided either the assumption D = E has been made, or the assumption has not been made and, making the assumption, E and the defining declarer of D are equivalent;
Step 3 if E is a mode-indication, then D and E are interchanged and Step 2 is taken;
Step 4 if D and E are declarators beginning with a reference-to-symbol (R7.1.1.l,m,n), then they are equivalent provided the declarers obtained from them by deleting the reference-to-symbol are equivalent;
Step 5 if D and E are declarators beginning with a procedure-symbol (R7.1.1.w), then they are equivalent provided all corresponding constituent virtual-parameters of the parameters-pack, if any, of their virtual-plan (R7.1.1.x,aa) are equivalent, and, furthermore, the virtual-declarers following their virtual-plan either are virtual-void-declarers, or are equivalent;
Step 6 if D and E are declarators beginning with a structure-symbol (R7.1.1.e), then they are equivalent provided in all corresponding constituent field-declarators the declarers are equivalent and the field-selectors are the same;
Step 7 if D and E are declarators beginning with a sub-symbol (R7.1.1.o,p), then they are equivalent provided they contain the same number of constituent upper- and lower-bounds, and the declarers following their bus-symbol are equivalent;
Step 8 if D and E are declarators beginning with a union-of-symbol (R7.1.1.cc), then they are equivalent provided to every constituent virtual-declarer of D there is an equivalent constituent virtual-declarer of E and vice versa;
Step 9 D and E are not equivalent.

4. Concluding remarks

The mode-declarations of a program can be seen as rules in a deterministic context free grammar, with the mode indications as nonterminals. The algorithm uses knowledge of the possible structure of the rules (declarers) to decide whether the (only) terminal production of two nonterminals are the same finite or infinite sentence. It is not customary to talk about infinite sentences but there seem to be no problems involved in this special case. It is, of course, in general undecidable whether the languages produced by two grammars are the same.

I will not prove that this algorithm in fact proves equivalence according to the definition given, but I will show that it always terminates. In performing the algorithm some number of assumptions are made, and only by going on making assumptions it could cycle. Every assumption is of the form M = D, where M is a mode indication contained in the program, and D is a declarer contained in the program. The number of such mode-indications and declarers is finite, and so consequently is the number of possible assumptions. No assumption is made twice as can be seen from Step 2; so the algorithm terminates.

The algorithm has one drawback, as can be seen from

E22) `mode m = m, n = real`

It will find m = n, in other words, it can not be finitely disproved that m and n specify one same mode. Indeed, m, which does not seem to specify any mode, is equivalent to every declarer. Luckily this type of paradoxical mode-declaration is forbidden by R4.4.4.c.

The algorithm given looks formidable; in fact it is very fast and efficient. A version of it has been programmed in ALGOL 60, and recognises in neglectable time the equivalence of

```
E23)  mode m1 = struct (ref struct (ref m2 t1, ref m3 t2) t1, ref m1 t2)
E24)  mode m2 = struct (ref m3 t1, ref struct (ref m1 t1, ref m2 t2) t2)
E25)  mode m3 = struct (ref struct (ref m2 t1, ref m1 t2) t1,
                       ref struct (ref m3 t1, ref m2 t2) t2)
```

The central procedure of the program is in ALGOL 60

```
E26)  Boolean procedure equivalent (d1, d2); integer d1,d2;
      if same sequence of symbols (d1, d2) then equivalent:= true else
      if mode indication (d1) then
      begin   if postulated (d1, d2) then equivalent:= true else
              begin   postulate (d1, d2);
                      equivalent:= equivalent (defining declarer (d1), d2)
              end
      end else
      if mode indication (d2) then equivalent:= equivalent (d2, d1) else
      equivalent:= same tree (d1, d2);
```

The boolean-procedure equivalent performs step 1, 2 and 3 of the algorithm, and delegates the remaining steps to same tree, which in turn relies heavily on equivalent.

References:

R Report on the Algorithmic Language ALGOL 68, version MR99 or MR100, A. van Wijngaarden (Editor).

K The Art of Computer Programming, Vol 1 / Fundamental Algorithms, 1968, D.E. Knuth.