AB42.4.5 Proposals for Algol H-a superlanguage of Algol 68

A.P. Black, V.J. Rayward-Smith
School of Computing Studies, University of East Anglia

§1. Introduction

This paper is devoted to the description of a proposed extension of Algol 68 which we shall call Algol H. The motivation for the development of this language comes from [1]. Sections 3 to 7 of this paper correspond to the sections in [1] with the same names. They describe constructions in Algol H which represent the abstractions of Hoare's theory.

Professor Hoare makes a vigorous disclaimer in the introduction to his paper: he is not embarking on the design of yet another programming language [1]. His abstract data structures are intended to assist in the formulation of abstract programs, and in their representation as concrete code. The transition from abstract to concrete is an essential part of the program design process, and there are good reasons why it should not be automated. Hoare draws a clear distinction between an algorithmic language and a programming language, his notations being an example of the former and Algol an example of the latter. However, Hoare admits there are advantages in the programming language being a subset of the algorithmic language. Many of his notations can be implemented with high efficiency; this is certainly true of unstructured data types and of elementary data structures, viz., Cartesian products, unions, arrays and powersets.

The advanced data structures, namely sequences, recursive structures and sparse structures, are fundamentally more difficult to implement by an automatic translator. Consequently, no proposals are made here to include these structures in Algol H (except in as far as transput is a representation of certain kinds of sequence).

Many of the new constructions proposed for Algol H have counterparts in the programming language Pascal [4,8]. Enumerations are available (always ordered), and so are subranges, although these are more restricted than the submodes of Algol H in that all subranges of a given parent type must be disjoint. The syntax of Pascal is markedly different from that of Algol H, which is intended to introduce these concepts to programmers more familiar with Algol 68. Some of the constructions of Algol H have previously been described as Algol 68 "might-have-beens" [7].

§2. The Method of Definition

As an Algol 68 superlanguage, Algol H should be described by a (two-level, Van Wijngaarden or) W-grammar, as is used in the Report. (Here, and in all that follows, "Report" means the Revised Report on Algol 68 [2]. References to specific sections are given as, e.g. [R 2.2.2.c].) However, this has certain disadvantages, for W-grammars have been held to be difficult to understand, particularly by those who do not know the language they describe. Whilst we feel this difficulty is often overestimated, it is none the less real. For this reason, the constructs of Algol H are described here only by means of examples and natural language. It is hoped that this will be "easier for the uninitiated reader", but for the initiated, part of the W-grammar definition of Algol H can be found in [3]. It should be noted that it is not easy to extend the grammar of Algol 68 so that it defines Algol H. For example, in Algol 68 the metanotion NEST, which carries a record of all the declarations forming the environment, has no need to envelop denotations, since the meaning of a denotation is independent of any nest [R.8.0.1]. In Algol H this is not so; it is possible to declare enumeration modes whose denotations are scoped.

§3. Unstructured Data Types

All data structures in a program must be built up from unstructured components. Most programming languages provide some unstructured types, usually reals and integer, and in theory these are adequate for all purposes. In practice there are strong reasons for defining other unstructured types. For example, although character values can be represented as integers, it has become usual to provide a character type in text processing programming languages. This has two advantages. First, the potential range of values of a variable is made explicit, thus making the program clearer and subject to more compile-time checks, which can detect such errors as the addition of an integer to a character. Second, it is possible to devise an efficient representation; because the cardinality of the character set is usually much less than that of the set of permitted integers, characters can be represented in less bits in the computer memory.

The next step, after including in a language a wider choice of basic modes, is to allow the programmer to define his own unstructured modes, either by enumeration of values or by taking a subrange of some existing mode. However this is done, it is fundamental that different data types should be represented as different modes. Since this mode is known at compile time it is possible to ensure that unrelated data are not mixed. The protection thus provided is one of the principal benefits of the extensions.

3.1 Enumerations

In many programs integers are used to denote a choice from a small number of alternatives rather than numeric quantities. In such cases we may expect the documentation of the program to list the possible values with their intended interpretation. For example, one might find the following in a program concerned with bidding sequences in bridge.

```
int trumps, bestsuit;
c  These variables refer to integers which
   indicate suit values as follows:

   0 - clubs
   1 - diamonds                                          (i)
   2 - hearts
   3 - spades
   4 - no trumps
c
```

The notion of an enumeration enables quantities such as suits of cards, sexes or colours to be represented as separate modes, quite distinct not only from the integers but also from each other. In Algol H the following are legal mode declarations.

```
mode suit = order (clubs, diamonds, hearts, spades, notrumps)
mode sex = scalar (male, female)
mode primary colour = scalar (red, green, blue)
mode dayofweek = order (Sunday, Monday, Tuesday, Wednesday,
                        Thursday, Friday, Saturday)
```

The use of the order-token (*order*) indicates that the mode should be considered to be ordered; the scalar-token (*scalar*) denies any such ordering. The bold-TAG-symbols (*clubs*, *male*, etc.) are MODE denotations for the various MODES, in exactly the way that, in Algol 68, *true* and *false* are boolean denotations. To be pedantic, the parallel is not exact, for *true* is the representation of the true-symbol not the bold-letter-t-letter-r-letter-u-letter-e-symbol. Nevertheless, it is clear [R.4.2.2b] that the similarity is intentional. This distinction does not arise since in Algol H *bool* is not a primitive mode but is defined using *mode bool* = *order (true, false)*. The use of the new denotations throughout the program enhances its understandability considerably; the assignation

```
trumps := spades
```

conveys more information (to the human reader) than

```
trumps := 3                                              (ii)
```

which would be its counterpart on the assumption of trumps being an *int* variable, as in example (i).

The situation can be alleviated in Algol 68 by use of identity declarations for appropriate integers:

```
int clubs = 0, diamonds = 1, hearts = 2, spades = 3, notrumps = 4
```

which may then be followed by

```
trumps := spades
```

in the same context as (ii).

This use of ascription is no substitute, however, for the ability to define new basic modes: the advantages of enumeration modes become clearer when control structures are introduced to manipulate them, but immediately we see that there can be better protection, more efficient store utilisation, and a closing of the gap between the data structures of a program and the real world objects they represent.

As far as protection is concerned, not only is the programmer less likely to assign objects of one type to variables of another (because of the mnemonic names), but such assignations are in any case syntax errors, since each type of object is represented by a distinct mode in the program.

The standard transput routines may, of course, be applied to enumeration modes - a facility not available in Pascal. The external forms of the values of such modes will in general be implementation dependent. It is suggested that where the bold-TAG-symbols used in the program text for denotations are represented by stropping, the characters transput should be those forming the corresponding TAG-symbol.

In Algol 68, declarers specify modes. A declarer is either a declarator, which explicitly constructs a mode, or an applied-mode-indication, which stands for some declarator by way of a mode-declaration [R.4.6]. Thus, to introduce new basic types, it is necessary to provide new declarators. Syntactically, this is done by providing a new descendent of the notion VIRACT-MOID-NEST-declarator [R.4.6.1a].

Algol 68 is quite specific about the equivalencing of modes. For example, the modes specified by the mode indications *a* and *b* in

```
mode a = union (int, real);
mode b = union (real, int)
```

are equivalent.

Similarly, it is intended that the modes specified by the declarators *scalar (red, blue, green)* and *scalar (red, green, blue)* be equivalent. On the other hand, the declarators

```
        order (morning, afternoon, evening)
        order (afternoon, evening, morning)
    and order (morning, noon, night)
```

all specify different modes (which cannot co-exist in the same reach, since applications of the denotations cannot be uniquely identified). In [3], the metanotion MODE is extended to include the metanotion BASIC as an additional alternative [R.1.2.1.A]. BASIC envelops all possible enumeration modes as its descendants. The mode equivalencing syntax then needs extending so that it can compare permuted scalar modes, for example, to detect the equivalence of *scalar (male, female)* and *scalar (female, male)*.

3.2 Subranges

Another common requirement is to deal with quantities which, though intrinsically of a basic type, will take only a limited range of values - a subrange. Clearly the parent type must be ordered. The bounds of the submode must be denotations, optionally preceded by a sign if the parent mode is integral.

```
        mode dayofmonth = sub (1:31);
        mode letter = sub ("a":"z")
```

The parent mode can be an enumeration mode; for example, in the reach of

```
        mode dayofweek = order (Sunday, Monday, Tuesday, Wednesday,
                                Thursday, Friday, Saturday)
```

the following is a valid mode declaration.

```
        mode workingday = sub (Monday:Friday)
```

A subrange may also be defined as the union, intersection or difference of a pair of subranges, provided they both have the same parent mode. These operations are represented by ∪, ∩ and \ with their usual set theoretic meanings.

It is not possible to declare a submode of a submode since the submode has no denotations. For example, *sub ("i":"n")* specifies a sub-character mode, not a sub-letter mode, because *"i"* and *"n"* are character denotations.

In Algol H, there is a widening coercion from a submode to a mode which is available in strong positions.

3.3 Manipulation

Hoare lists seven operations required for the manipulation of values of enumeration and subrange modes.

Test of equality: The equality and inequality operators are defined for all enumeration modes, and all subrange modes which are not submodes of real.

Assignation: The assignation of values to names is exactly as in Algol 68.

Case discrimination: Algol H has a choice-clause which is a generalisation of the choice-clause of Algol 68. The advantages of enumeration modes become obvious when we compare equivalent Algol 68 and Algol H. The Algol 68

```
case trumps +1
in   20 × bid, 20 × bid, 30 × bid, 30 × bid
,    40 + 30 × (bid-1)
esac
```

is almost incomprehensible without the comment of Section 3.1, and is error prone even with it. In Algol H we can similarly write (because *suit* is ordered)

```
case trumps
in   20 × bid, 20 × bid, 30 × bid, 30 × bid
,    40 + 30 × (bid-1)
esac
```

but without having to perform arithmetic on *trumps*. The v^{th} constituent units of the in-choice-clause is elaborated when trumps takes the v^{th} value of the ordered enumeration. If there are less units than values in the mode, then the out-choice-part is elaborated for the extra values.

Alternatively, the constituent units can be prefixed with specifications, as in the following.

```
case trumps
in   (clubs, diamonds): 20 × bid
,    (notrumps): 40 + 30 × (bid-1)
out  30 × bid
esac
```

Each value of the mode may be mentioned in at most one specification; if the enquiry clause takes a value not mentioned in any specification, then the out-choice-part is chosen. (If there is no out-part, a skip is elaborated, which may yield an undefined value.) If the MODE of the enquiry clause is ordered then sub-of-MODE-declarators are permitted as specificators; for example, given the declaration *mode letter* = *sub* *("a":"z")* we write a skeleton scanner.

```
case ch := readch
in   (letter) :    serial clause
,    (sub("0":"9"),".") :    serial clause
,    ("""") :    serial clause
,    ("$") :    serial clause
,    ("'") :    serial clause
out   serial clause
esac
```

If, on the other hand, the MODE of the enquiry clause is not ordered, then the form of the choice clause which does not use specifications is not permitted, since there is no implicit ordering of the values of the enumeration which can be used to choose the appropriate unit from the in-choice part.

Ordering relations: The dyadic operators <, <=, >, >= are declared in the standard prelude between operands of the same mode and yielding boolean results. They are defined for a sufficient set of ordered modes, each member of which has order.

Counting: The monadic operators *succ* and *pred*, invoking the successor and predecessor functions, are defined for all simple, ordered modes. These functions map each value of a mode onto the next higher or lower value (if there is one, otherwise their action is undefined).

Also, for each ordered mode for which there is a mode declaration in a given reach, if the mode indication defined by that declaration is some bold-TAG-symbol, then the identifiers max-cum-TAG-symbol and min-cum-TAG-symbol are declared and ascribed the maximum and minimum values of the mode. The example should make this clear.

```
mode rank = order (private, corporal, sergeant,
     lieutenant, captain, major, colonel, general);
c    The effect on the nest is as if, in place of
     this comment, there stood the declaration
          rank maxrank = general, minrank = private
c
...
```

The loop clause: It is frequently required to elaborate the same serial clause for all possible values of a given mode. This is indicated by the use of a loop-clause.

```
loop dayofweek day
do
      dailytask (day)
od
```

The loop-symbol is followed by a formal-declarer which gives the mode of the identifier whose value is varied. If the mode is ordered and has cardinality n, say, then the serial-clause between *do* and *od* is elaborated n times in sequence as the identifier takes, in order, the values of the mode from minmode to maxmode. However, if the mode is not ordered, the effect is as if n copies of the serial clause were elaborated collaterally. The loop clause is available for all modes derived from SIMPLE, not just for integral as in Algol 68. However, the special *for from by to* construction of Algol 68 is retained for use by integers, since conceptually that mode has infinite cardinality.

Transfer functions: It is sometimes required to perform operations for a submode which were defined for the parent mode. This is simply accomplished by converting the submode value to the corresponding value of the larger type, then performing the operation, and finally converting back again if necessary. This requires transfer functions.

The first conversion is a widening, and can be accomplished by the coercion described in Section 3.2. If the context is not strong enough (as will be the case if the coercee is the operand of a formula) the coercion can be forced by the use of a cast.

```
mode smallint = sub (-9:9);
smallint si ,    sj ,    sk; int k
...
k := int (si) + int (sj)
```

It is, of course, possible to declare a version of + between small int operands, but this must be done explicitly by the programmer.

The second conversion represents a narrowing, and can be accomplished only with the aid of a standard operator.

```
sk := smallintval k;
```

This operator is automatically declared in any reach in which a mode declaration is given for a submode. The letters of the TAG used for the operator are given by concatenating the letters of the bold-TAG-symbol defined in the mode declaration with letter-v-letter-a-letter-l.

If this operator is applied to an operand whose value is outside the range encompassed by the submode, then the result is undefined. It is therefore useful to be able to check if a given value is in a subrange. This can be done by a conformity relation, which is based on the conformity relation for unions which was part of the language defined in 1968 [6]. The symbols represented by *::=* and ctab denote a conform-to-and-becomes relation, whereby the object on the right hand side is assigned to the variable on the left (and true delivered) if it is in range, otherwise no assignation takes place (and false is delivered). The symbols represented by *::* and ct invoke the same test but without the assignation. The use of these relations is illustrated in the following examples.

```
char c; letter a; digit b;
read (c);
if a ctab c then c an assignation to a has just occurred c
                   serial clause
elif b ctab c then c an assignation to b has just occurred c      (i)
                   serial clause
else c neither a nor b have changed c
     serial clause
fi
if letter ct c then c c is in the subrange letter c
                     serial clause                                 (ii)
fi
```

In the second example, the conformity relation has the property that its left hand side is not elaborated, i.e. no space is generated on the heap. The generator letter is simply there for mode matching. Note that ct can be used even when no mode declaration has appeared for a submode, and consequently it is not identified by a bold-TAG-symbol, so no submode-cum-val operator is defined.

Very similar results could instead be achieved by standard operators, also identified by ct and ctab. Such operators would be defined between a sufficient set of modes as right operand, and a sufficient set of submodes of each mode as left operand. Such operators would have the disadvantage that both operands would always be evaluated; in the second example above, space for a letter value would be generated on the heap, and immediately become garbage.

§4. The Cartesian Product

The Cartesian product is one of the simplest data structures. It corresponds to the record structures of PL/I [9] and Pascal or the Algol 68 structured modes. In fact, with one exception, Algol 68 structures provide all the facilities suggested by Hoare [1], although the notation is different. In particular, values of a structured mode are constructed, in Algol 68, from a collateral; where the required mode is not obvious from the context, a cast of the collateral is used. This corresponds to Hoare's suggestion that it would be convenient to leave the transfer function implicit in cases where no ambiguity would arise.

The exception is the *with* construction. In inspecting or processing a structured value, it is often required to make many references to its components within a single clause. Hoare favours a special construction which could be represented as

with structure closed clause .

Within the closed clause the fields of the structure may be referred to by their field selectors alone, instead of by the normal selector *of* primary construction. It is debatable whether the advantages of this construction, viz., the clarification and abbreviation of a section of program, are great enough to outweigh the disadvantage of introducing another construct into the language. Algol 68 is often called a complex language, and ascription provides most of the power of the with clause. For these reasons, it is proposed that Algol H does not include this construction.

§5. The Discriminated Union

Although similar to Algol 68 union, Hoare's Discriminated Unions differ in certain respects. Each type in the union has an identifier associated with it, and every value of one of the unioned types is marked with its identifier, to indicate its derivation. Thus it is possible to construct a union by repeating the same type several times.

In Algol 68, in contrast, the declarator,

union (date, date)

is ill-formed. The intention that Algol H should be a superlanguage of Algol 68 dictates that the Algol 68 kind of union be included in Algol H.

Since the advantages of discriminated unions are debatable, and it would in any case be confusing for one language to provide two very similar but distinct structures, discriminated unions are not included in Algol H.

§6. The Array

The array is a very familiar data structure, and may be regarded as a mapping from a domain of one type to a range of a possibly different type (the type of the array elements).

In most programming languages, the most notable exception being Pascal, the domain type is restricted to be integral. This is true of Algol 68. Algol H allows more general arrays; the mode of the domain can be any enumeration mode, SIZETY integral, character or a submode of any permitted mode.

Some examples should make this clearer.

```
[suit] int trickvalue;
[day of week] bool holiday;
[sub (1:80)] char punchcard
```

Elements of these arrays can be selected in the usual way.

```
day of week today := sunday;
if not holiday [today] then work else sleep fi
```

The facility of using a subrange of integers as the domain mode does not replace the ordinary Algol 68 row with integer indices, because the limits of a submode can only be plus or minus a denotation, and it is therefore not possible to read in submode bounds at run-time.

§7. The Powerset

The powerset of a set is the set of all subsets of that set. The powerset as a data type takes values which are sets of values selected from some other data type known as the base. *Primarycolour* has been declared as an enumeration mode with cardinality 3; the powerset of primarycolour is a mode which has a cardinality of $2^3 = 8$. Its values are as follows.

```
{ }        {red, green, blue}
{red}      {red, green}
{green}    {red, blue}
{blue}     {green, blue}
```

Declarators for powerset modes take the form illustrated by

```
setof primarycolour
```

so we may construct the mode declaration

```
mode colour = setof primarycolour
```

and then declare some identifiers

```
colour yellow = {red, green},
         cyan = {blue, green},
         blue = {blue},
        black = { }
```

The object between and including the braces is a powerset clause. The values within it must all be of the same mode, but their order is immaterial and a repeated value is ignored, so that {*green*, *red*, *red*} represents the same value as {*red*, *green*} which would not be the case for a collateral clause. The empty powerset clause { } can only stand in a strong position. An alternative representation of the braces are the symbols *set* and *tes*.

7.1 Manipulation

For each powerset for which a mode declaration is given, a constant is declared corresponding to the universal set, where this has finite cardinality. In the example, the effect is as if the declaration

colour *allcolour* = {*red*, *blue*, *green*}

were elaborated after the mode declaration.

Where the cardinality of the base type is small the powerset can be efficiently represented by allocating one bit in store for each value of the base type. Thus, for example, values of type colour can be represented in three bits. The basic operations on powersets are usually available as single machine instructions, which makes this representation doubly attractive. However, when it is known that the cardinality of the base type is large, or perhaps conceptually infinite, the bit pattern representation loses its attraction, particularly when most values of the powerset will consist of only a small number of elements of the base. In these cases it will be necessary to represent the powerset as some advanced data structure, which an automatic translator cannot be expected to construct.

However, the utility and efficiency of the powerset of a small base type is such that it ought to be included in Algol H; powersets are therefore permitted providing the cardinality of the base type does not exceed some implementation defined maximum possessed by the standard prelude integer *setwidth*.

Various operations can be defined between sets of a given powerset mode and elements of its base mode. There is one operator defined between a set *s* and an element *x*.

Membership : *x* *isin* *s* delivers *true* if the element
x is a member of set *s*

The following operators, with their usual mathematical meaning, are defined between two sets:

equality,
union,
intersection,
relative complement,
inclusion.

Algol H also allows assigning versions of union, intersection and relative complement.

Union and becomes	: $s1 \cup := s2$	equivalent to	$s1 := s1 \cup s2$
Intersection and becomes	: $s1 \cap := s2$	equivalent to	$s1 := s1 \cap s2$
Difference and becomes	: $s1 \setminus := s2$	equivalent to	$s1 := s1 \setminus s2$

It is also useful to be able to select an element from a set, and simultaneously remove it. This is achieved by the operator *outof*: x *outof* s removes an element from s and assigns its value to x.

§8. Implementing Algol H

A major part of the work of any parser for Algol 68 is to perform mode-checking. Because, in general, a given mode can be "spelled" in a large (sometimes infinite) number of ways, the syntax of Algol 68 includes complex devices which check if modes are equivalent and if a value of a given mode is "acceptable" to another mode [R.2.1.3.6, 2.1.4.1]. The latter test depends on the syntactic position (SORT) of the construction, as the list of applicable coercions vary with context. All this must be modelled within a parser, which is no small task.

However, that this task can be successfully completed is evidenced by the existence of several usable and efficient Algol 68 compilers. Given access to such a compiler and a description of its mode representation mechanism, one feels that it would not be too difficult to extend it to encompass new basic modes.

This feeling is reinforced by the comments of Wirth on the programming languages Pascal and Modula. Pascal has scalar types which correspond to the ordered enumeration modes of Algol H, and subrange types which provide some of the facilities of submodes. Arrays with general domains and powersets are also included. Nevertheless, Wirth states that one of his principal aims in developing Pascal was to produce a language which could be implemented reliably and efficiently, both at compiler and execution time. In [10] he writes "a most important consideration in the design of Modula was its efficient implementability". Modula is a language based on Pascal but with an improved syntax and the ability to associate access procedures with data in the manner of a Simula [11] class; it includes enumerations, which may form the indices of arrays, and is intended for use in real-time applications on mini-computers.

Thus there is justification for being confident that most of the new constructions of Algol H could be efficiently implemented by an extensible Algol 68 compiler. To the best of our knowledge no such compiler exists; we have in mind an extension device akin to the Syntax Macros of Cheatham [12] and Leavenworth [13]. In [3], a translator from Algol H to Algol 68-R [14] is described where each Algol H construct is translated into Algol 68-R in such a way that all mode checking is done by the Algol 68-R compiler.

Acknowledgement

The authors wish to thank Professor R.J.W. Housden for helpful discussions. The majority of this work was done by A.P. Black under the supervision of Dr. Rayward-Smith in partial fulfillment of the requirements of the B.Sc. degree at the University of East Anglia, Norwich.

References

[1] C.A.R. Hoare, "Notes on Data Structuring" in "Structured Programming" by O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press (1972).

[2] A. van Wijngaarden, et al., "Revised Report on the Algorithmic Language Algol 68", Acta Informatica, 5 (1975).

[3] A.P. Black, "Algol H: Some Extensions to the Data Handling Facilities of Algol 68", Project Report, School of Computing Studies, University of East Anglia, Norwich (1977).

[4] N. Wirth, "The Programming Language Pascal", Acta Informatica, 1 (1971).

[5] C.H. Lindsey and S.G. van der Meulen, "Informal Introduction to Algol 68", revised edition, North Holland (1977).

[6] A. van Wijngaarden, et al., "Report on the Algorithmic Language Algol 68", Numerische Mathematik, 14 (1969).

[7] S.G. van der Meulen, "Algol 68 Might-Have-Beens", Proceedings of the Strathclyde Algol 68 Conference, Sigplan Notices, 12:6 (1977).

[8] K. Jensen and N. Wirth, "PASCAL: User Manual and Report", Lecture Notes in Computer Science, 18, Springer Verlag (1975).

[9] C.P. Lecht, "The Programmers' PL/1", McGraw Hill (1968).

[10] N. Wirth, "Modula: A language for Modular Multiprogramming", Software-Practice and Experience, 7:1 (1977).

[11] G.M. Birtwistle, O.J. Dahl, B. Myhrhang, K. Nygaard, "Simula Begin" Auerback (1973).

[12] T.E. Cheatham Jnr., 'The Introduction of Definitional Facilities into Higher Level Programming Languages", Proc AFIPS FJCC, 29 (1966).

[13] B.M. Leavenworth, "Syntax Macros and Extended Translation", CACM, 9:11 (1966).

[14] P.M. Woodward, S.G. Bond, "Algol 68-R Users Guide", H.M.S.O. (1974).