stichting
mathematisch
centrum

MC

AFDELING INFORMATICA (DEPARTMENT OF COMPUTER SCIENCE) IW 73/77 JANUARI

J.C. VAN VLIET

TOWARDS A MACHINE-INDEPENDENT TRANSPUT SECTION

Prepublication

2e boerhaavestraat 49 amsterdam

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

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A30

ACM-Computing Reviews-categories: 4.22, 4.12, 4.41

Towards a machine-independent transput section [*)]

by

J.C. van Vliet

ABSTRACT

If the transput section of an ALGOL-68 compiler is to be portable, it must be described in such a way that it is clear which aspects are machine-dependent, and which are not. There should be a clear set of primitives underlying the transput. In this report, a description is proposed which can really be used as an implementation model: the transput is described in pseudo-ALGOL 68, except for the underlying primitives, whose semantics are given in some kind of formalized English. The state of this model is by no means definitive, but may serve as a start for further discussion.

KEY WORDS & PHRASES: *ALGOL 68, transput, portability, implementation model, run-time system, file system.*

---

*) This report will be submitted for publication elsewhere.

INTRODUCTION

In the Revised Report on the Algorithmic Language ALGOL 68 [1] (henceforward referred to as the Report), Chapter 10 deals with the standard environment. This standard environment is described in pseudo-ALGOL 68 and comprises two main components: a collection of (mainly mathematical) functions and operations, and the transput (i.e., input and output). The standard environment is part of the run-time system of an ALGOL-68 compiler.

One of the main objectives of the ALGOL-68 compiler that is being developed at the Mathematical Centre is portability. Several aspects can be distinguished:

- for a compiler to be portable, the language in which the compiler is written must be portable;
- there should be a clear interface with the machine on which the language is implemented; this interface should be independent of that specific machine;
- the run-time system should be easy transportable.

The run-time system of an ALGOL-68 compiler is likely to be a very substantial part of that compiler, so its portability will heavily influence the portability of the compiler as a whole. Apart from the standard environment, the run-time system mainly contains storage management routines (garbage collector!). These interact strongly, and both depend on the machine-independent object code chosen; they will not be discussed in this paper. This paper, and the one by D. GRUNE [2], will focuss on the other aspect of the run-time system: the standard environment.

If the standard environment is to be portable, it must be described in such a way that it is clear which aspects are machine-dependent, and which are not. There should be a clearly defined set of primitives underlying the standard environment, and this set should in some sense be small. These primitives will then form the operating-system interface. The "meaning" of these primitives must also be defined. Defining their meaning might well turn out to be as difficult as determining them. As WAITE [3] states: "If the meaning of the entire program is to remain invariant, the meaning of the invariants must remain invariant".

The approach we have chosen is to describe the standard environment in pseudo-ALGOL 68. The pseudo part is then to be considered as a language extension which should be reasonably implementable. This pseudo part is described by D. GRUNE [2]. The underlying primitives of our model are not defined in ALGOL 68. Instead, their semantics are given in some formalized English, resembling the way in which the semantics in the Report are defined. One advantage of a description in pseudo-ALGOL 68 is that it can largely be tested mechanically. It also becomes more portable, especially because the pseudo-ALGOL 68 part is accepted by our compiler too.

The transput section of the Report offers little or no help in finding the underlying primitives. It may only be looked upon as a description of the *intention* of transput. To give one obvious example: no implementer will seriously consider the possibility of implementing the text of a file as [ ][ ][ ] *char*, if his operating system already takes a different view on files. As a consequence, the burden of finding all tricky spots is placed upon the shoulders of each individual implementer. This effectively means that the transput has to be rewritten for each implementation. The situation becomes muddled if we take into account that certain ALGOL-68 limitations have clearly influenced the description. For example, the mode *format* almost exactly mirrors the structure of format texts. However, due to scope restrictions, for the level above collection a different structure is used.

The advantages of a transput section that can be used as an implementation model are obvious. Not only could it help to make it unnecessary to re-implement it from the very beginning, but it could also be a means to arrive at standardization of the transput. (Of course, no matter how careful we are, there will always be operating systems that do not support some of the primitives that are assumed available.) Existing implementations offer little or no help with respect to standardization. The diversity, with regard to transput, is striking:

- The ALGOL 68R implementation [4] offers some version of the transput. However, this is not the transput as described at any stage by an ALGOL 68 Report, but it is some locally developed system;
- The Control Data implementation [5] covers almost exactly the transput described by the Report. But it is a commercial product, and its internal structure is proprietary information;
- In the Munich implementation [6], the transput is dealt with in a

machine-dependent way;

- Although it is not stated explicitly, it appears from [7] that the Brussels group has implemented some version of the transput. In general, they tended to stay as close to the original report as possible (whose transput differs greatly from that of the Revised Report);
- The Oklahoma implementation [8] deals with transput in a largely interpretive way. In [9], an interpreter for formatted transput, as described in the original report, is given. This interpreter is written in FORTRAN;
- The Rennes implementation [10] contains a form of transput which is tailored to their SIRIS 7 operating system;
- In the ALGOL 68 Student compiler [11], almost the whole transput is implemented, although they changed it slightly to make it fit their own ideas;
- In general, sublanguages of ALGOL 68, such as ALGOL 68C [12], ALGOL 68S [13], ALGOL 68/19 [14] and Mini ALGOL 68 [15], have very limited transput capabilities. Usually, the first thing to be dropped is formatted transput.

It is clear from the above that there is not much uniformity. Very likely, each implementation has a different view on transput and implements something different. This tendency should be counteracted! Let us therefore start a discussion towards an agreement on a standard transput, similar to the way in which we agreed upon a standard representation [16]. If such a standard would be available in the form of an implementation model, it would greatly alleviate the task of the individual implementer, and in this way (indirectly) stimulate the diffusion of ALGOL 68.

In the following sections, two examples of parts of the ALGOL-68 transput are discussed, and an attempt is made to give a clear definition of the underlying primitives. The first example deals with the routine *float* (section 10.3.2.1.d of the Report). It is shown that one clearly defined primitive suffices for the conversion of real numbers to strings. This example is taken from [17], where the transput conversion routines are treated in full detail. The other example deals with books, channels and files (section 10.3.1. of the Report). It is important to note that the current state of our model is by no means definitive, but may serve as a start for further discussion.

CONVERSION BY MEANS OF FLOAT

The main problem in converting real numbers to strings is a numerical one. An important application of the conversion routines is to measure the accuracy of numerical algorithms, and we want to ensure that that is really what is measured, and not the accuracy of the conversion. It is obvious that if this conversion is completely machine independent, it can not be accurate.

In the set of conversion routines that is proposed in [17], numbers are first converted to strings of sufficient length (this part is machine dependent), after which all arithmetic is performed on these strings. This version may indeed be seen as an implementation model: for each direction of conversion, there is only one place where real arithmetic comes in. As an illustration, the routine *float* will be treated in greater detail.

The routine *float* is intended to convert real numbers to floating point form. It has four parameters:

- *v*, the value to be converted,
- *width*, whose absolute value specifies the length of the string that is produced;
- *after*, whose value specifies the number of digits required after the decimal point, and
- *exp*, whose absolute value specifies the width of the exponent.

A sign is normally included in both the mantissa and the exponent. The user may specify that a sign is to be included for negative values only by supplying a negative width. If the value of the *exp* parameter is zero, *float* acts as if minus one were specified, i.e., the exponent is converted to a string of minimal length. The value of the *width* parameter, however, may not be zero.

From the routine given below, the following may be observed:

(i) The routine does not use real arithmetic. Unless the exponent is of the order of magnitude of *max int*, which is very unlikely, the integer arithmetic neither presents any trouble;

(ii) The routine does not distinguish between various lengths of numbers; they are just passed down to *subfixed*;

(iii) Numbers are first converted to strings of sufficient length, after which the rounding is performed on the strings. This seems to be the only reasonable way to ensure that numbers like *L max real* may be converted using *fixed* or *float*;

(iv) The routine is written non-recursively;

(v) Care has been taken that the routine behaves exactly as the corresponding routine from the Report was probably intended to. This will not be completely true for the routine *subfixed*, which necessitates some changes in the editing of integers and reals in the routine *putf* in section 10.3.5.1. of the Report.

(These remarks apply equally well to the routine *fixed* in [17], where also an extensive description of the working of the conversion routines is given.)

```
proc float = (number v, int width, after, exp) string:
  begin int before := abs width - (after ≠ 0 | after + 1 | 0) - (abs exp + 1),
            exponent, aft := after, exspace:= abs exp;
    bool neg, rounded := false, possible:= true;
    string s:= subfixed(v, before + after, exponent, neg, true), expart:= "";
    (neg ∨ width > 0 | before -:= 1); exponent -:= before;
    while expart:= (exponent < 0 | "-" |: exp > 0 | "+" | "") +
                             subwhole(abs exponent, loc bool);
      if sign before + sign aft ≤ 0
      then possible:= false
      elif upb expart > expspace
      then expspace +:= 1;
        (aft > 0 | aft -:= 1;
          (aft = 0 | before +:= 1; exponent -:= 1)
          | before -:= 1; exponent +:= 1); true
        elif rounded then false
        elif round(before + aft, s)
        then exponent +:= 1; rounded:= true
        else false
        fi
    do skip od;
    if ¬ possible then undefined; abs width * errorchar
    else (neg | "-" |: width > 0 | "+" | "") + s [: before] +
         (aft = 0 | "" | "." + s[before + 1 : before + aft]) +
         "₁₀" + (expspace - upb expart) * "_" + expart
    fi
   end;
```

The routine *subfixed* performs the actual conversion from numbers to strings, and may be called from either *fixed* or *float*. When called from *fixed*, it must return a string containing all digits from the integral part of the value submitted, and *after* + *1* digits from the fractional part. When called from *float*, it must return a string containing the first *after* + *1* significant digits. In both cases, the last digit is truncated, and not rounded. (The rounding is done later on, and rounding the number twice may cause something like *9.46* to be converted to *"10.0"*.) Considering this string as a number, the value of the parameter *p* will be the shift of the decimal point from the first digit. The parameter *neg* will indicate the sign of the value submitted (true iff negative).

The routine *subfixed* must be completely accurate: As we said before, it will be used to measure the accuracy of numerical algorithms, and we do not want these measurements to be downgraded by the inaccuracy of the conversion. It is therefore impossible to give an ALGOL-68 routine that will do. Instead, we give the following semantic definition:

It is a unit which, given a value V, yields a value S and makes *p* and *neg* refer to values P and B, respectively, such that:

- B is true if V is negative, and false otherwise;
- it maximizes

$$M = \sum_{i = \underline{\text{lwb}}\ S}^{\underline{\text{upb}}\ S} c_i * 10^{P-i}$$

under the following constraints:

- lwb S = 1;
- upb S = P + *after* + 1 if *floating* is false, and *after* + 1 otherwise;
- for all i from lwb S to upb S:
  $0 \le c_i \le 9$, where c_i = *char dig*(S[i]);
- $M \le |V|$.

Remark: So that one need not know the storage allocation techniques used by the compiler (which are needed to build the string), one may construct an embedding like:

```
proc ? subfixed = (number v, int after, ref int p, ref bool neg, bool floating)
        string:
  begin int size; guess storage(v, after, size, floating);
        # size:= some sufficiently large integer, an upperbound for
          the number of digits that will result #
    [1 : size] char s;
    do subfixed(v, after, p, neg, floating, size, s);
        # the actual conversion; the characters are placed in s.
          As a side-effect, size indicates the number of digits placed
          in s #
    s[ : size]
  end;
```

End of remark.

The (hidden) routine *round* is used for rounding. The parameter *s* refers to the string to be rounded, the parameter *k* is the index of the last element of *s* that will be returned. The routine yields true if the rounding causes a carry out of the leftmost digit.

```
proc ? round  = (int k, ref string s) bool:
  if bool carry:= char dig(s[k + 1]) ≥ 5; s:= s[ : k]; carry
  then
    for j from k by -1 to 1 while carry
```

```
      do int d = char dig(s[j]) + 1; carry:= d = 10;
        s[j]:= (carry | "0" | dig char(d))
      od;
      (carry | "1" plusto s); carry
    else false
    fi;
```

BOOKS, CHANNELS AND FILES

Books, channels and files model the actual transput devices. Therefore, it is to be expected that most of the machine-dependencies of the transput section are located in this area. The status of what is discussed below is much more premature than that of the example given before. This part of our system is still changing every week. Therefore, the discussion will be rather informal, and certainly no formal semantics of any primitive can be expected here. Still, some fundamentals are already visible, and deserve to be discussed.

Let us start with the concept of a book. In my opinion, a book may best be seen as modelling the actual device. As such, quite a few machine-dependencies may be expected. Indeed, nothing is specified in our model about the internal structure of the mode *book*; as such, none of its fields are accessed. If something is needed, procedures are provided that will yield the information asked for; *books* are only used as parameters to procedures. More specific, our considerations regarding the various fields of the mode *book* from the Report are the following:

- The *text* is expected to be stored away in some opaque way; only parts thereof are available at each instant of time;
- The logical end of the book will in general not be known: it is only recognized as such when it is reached;
- The identification, whether in the form of a string or not, is expected to be very operating-system dependent;
- The other two fields, *users* and *putting*, are used to answer the question "Can I write on this book or not", which is also best answered by the operating system.

In our view, a channel is a collection of attributes that is common to some set of devices. As such, it may be considered quite independent of any specific machine. However, it can still be disputed whether or not the <u>*proc*</u> <u>*pos*</u> *max pos* is appropriate in the channel. It is only needed for channels on which files may be established. For those files, the maximum size must be known; this possibly also depends on the book.

Associated with the channel is the standard conversion key. It seems appropriate to link this conversion key to the channel. However, we do not specify the internal structure of this key, nor do we provide any conversion key. Of course, table-driven keys will likely be the fastest. (It must be noted that in various papers about system performance (see, e.g., WICHMANN [19]), the use of conversion keys is discouraged, because of it being too expensive.) So we do not specify the mode <u>*conv*</u>, and two conversion routines are supposed to be available: *convert int to ext* and *convert ext to int*. Two other machine-dependent routines associated with channels are *file available* and *match*.

The concept of a file, which is actually the file control, is the most heavily used concept in the transput section. In our model, it is considered to be largely machine-independent. In the Report, the file contains a direct reference to the text (apart from the indirect reference via the book), which contains the actual data of the file. As in the case of the book, we do not assume that the whole text is available at any instant of time. On the other hand, writing or reading each character separately to or from an external medium might well be very expensive, if not virtually impossible. Moreover, ALGOL 68 requires the ability to undo operations performed on the current line. This will probably not be possible if the character has already been punched. Therefore, it seems reasonable to take one line of the text (a buffer of one line) as a field of the file, and leave the rest of the text invisible. Other lines may then only be reached by means of calls of one of the routines *read line* and *write line*. (If, while writing, lines must be compressed, it is quite natural to delegate this to *write line* as well.)

This model suggests a *set* routine in which the new position is searched for by means of successive calls of *read line*. We may, however, want to provide a faster *set* routine for random-access files. If different *set* routines are available for different channels, it is quite natural to enter the *set* routine in the channel. In this way, it is unnecessary to search for the appropriate *set* routine, but it can simply be selected from the channel. (See, e.g., STOY & STRACHEY [20], for similar applications of this important idea.) It is probably advantageous to enter some of the other routines, such as *open*, in the channel too.

Another aspect of transput is the extensive testing that is done before a character is actually read or written. We have tried to concentrate some answers to these tests in the "status information" of the file, and provide fast routines to inspect this status. At this moment, the status of the file contains the following information:

- whether or not the file has been opened;
- whether or not the line has been ended;
- whether or not the page has been ended;
- whether or not the physical file has been ended;
- whether or not the logical file has been ended.

Routines *opened*, *line ended*, *page ended*, *physical file ended* and *logical file ended* are provided to inspect this status. After each transput operation, this status has to be updated. Updating *page ended* and *physical file ended* obviously has to take place behind the curtain. Routines *close file*, *line end* and *logical file end* are provided to update the appropriate information. To short-circuit the chains of tests that are activated upon calls of routines like *get good line*, the routines *char ok*, *line ok*, *page ok* and *logical file ok* are proposed, to yield quick answers to the corresponding questions. If such a routine yields false, the normal chain of tests is performed; otherwise, actual transput may continue.

The mode file thus gets the form:

```
mode file = struct(
               ref book book,
               channel chan,
               ref format format,
               ref line line,
               ref bool read mood, write mood, char mood, bin mood,
               ref pos cpos, ref int c of lpos,
string term,
conv conv,
ref status status,
ref int char bound,
proc (ref file) bool logical file mended,
                     physical file mended,
                     page mended, line mended,
                     format mended, value error mended,
proc (ref file, ref char) bool char error mended).
```

Note that *c of lpos* is embodied in the file. The only reasonable question concerning the logical end of the file is: "Have I reached the logical end of the file or not". The answer to this question is known after we filled the buffer with the line containing the logical end. For the moment, *c of lpos* is expected to have a value greater than the length of the current line if the logical end is not within the current line. Questions concerning the logical end will then automatically be answered negatively. Also, the maximum length of the current line is put in the file (*char bound*). Finally, the mode *format* is a real tree in our model, matching the actual format. Therefore, we only need a reference to the root of the format that is used at this moment.

Associated with the file are a number of machine-dependent enquiries. Apart from the actual shape of the mode *status*, the following routines are not (completely) specified in our model (;they are considered primitive, and their specification is to form the interface with the operating system):

- *char ok, line ok, page ok, logical file ok*;
- *opened, line ended, page ended, physical file ended, logical file ended*;
- *close file, line end, logical file end*;
- *read line, write line*;
- (part of) *close, lock* and *scratch*;
- (part of) *set* and *reset*;
- (part of) *establish, create, open* and *associate*;
- *reidf*;
- *undefined*.

This list is not meant to be exhaustive or final. However, we do think that most machine-dependencies that arise from the transput section have passed in review. It is hoped that further discussion may help us in reaching some consensus.

REFERENCES

[1] WIJNGAARDEN, A. VAN, et al (eds.), *Revised Report on the Algorithmic Language ALGOL 68*, Acta Informatica 5 (1975) 1-236.

[2] GRUNE, D.,*Towards the design of a Super-Language for the ALGOL 68 Standard Prelude*, IW 76/77, Mathematical Centre, Amsterdam.

[3] WAITE, W.M., *Theory*, in *Software Portability Course*, University of Kent at Canterbury, March 29 - April 9, 1976.

[4] WOODWARD, P.M. & S.G. BOND, *ALGOL 68-R Users Guide*, Royal Radar Establishment, Malvern, England, 1975.

[5] ALGOL 68 Version I Reference Manual, Control Data Services B.V., Rijswijk, The Netherlands, 1976.

[6] HILL, U., H. SCHEIDIG & H. WOESSNER, *An ALGOL 68 Compiler*, Technische Universität München and University of British Columbia, 1972.

[7] BRANQUART, P., et al, *An Optimized Translation Process and its Application to ALGOL 68*, Lecture Notes in Computer Science 38, Springer-Verlag, Berlin, 1976.

[8] ROBERTSON, A. & G.E. HEDRICK, *A Portable Compiler for an ALGOL 68 Subset*, in [18], p. 59-64.

[9] BERRY, R.D., *A Practical Implementation of Formatted Transput in ALGOL 68*, MS Thesis, Oklahoma State University, Stillwater, Oklahoma, July 1973.

[10] LEROY, A., et al, *On the Adequacy of the ALGOL 68 Environment Compared with an Existing Current Operating System and Problems of I/O Implementation*, in [18], p. 202-220.

[11] BROUGHTON, C.G. & C.M. THOMPSON, *Aspects of Implementing an ALGOL 68 Student Compiler*, in [18], p. 23-38.

[12] BOURNE, S.R., A.D. BIRRELL & I. WALKER, *ALGOL 68C Reference Manual*, University of Cambridge, Cambridge, England, July 1975.

[13] HIBBARD, P.G., *A Proposed Sublanguage of ALGOL 68*, ALGOL Bulletin 37 (July 1974), p. 30-54.

[14] GENNART, P.E., *Implementation and Usage of the ALGOL 68/19 Compiler*, in [18], p. 13-16.

[15] AMMERAAL, L., *Mini ALGOL 68 User's Guide*, IW 32/75, Mathematical Centre, Amsterdam.

[16] HANSEN, W.J. & H.J. BOOM, *Report on the Standard Hardware Representation for ALGOL 68*, ALGOL Bulletin 40, August 1976, p. 24-43.

[17] VLIET, J.C. VAN, *On the ALGOL 68 Transput Conversion Routines*, IW 61/76, Mathematical Centre, Amsterdam.

[18] HEDRICK, G.E., (ed.), *Proceedings of the 1975 International Conference on ALGOL 68*, Oklahoma State University, Stillwater, Oklahoma, June 10-12, 1975.

[19] WICHMANN, B.A., *Performance considerations*, in *Software Portability Course*, University of Kent at Canterbury, March 26 - April 9, 1976.

[20] STOY, J.E. & C. STRACHEY, *OS6- An Experimental Operating system for a small computer*. Part 2: *Input/Output and Filing system*, The Computer Journal 15 (1972) 195-203.