EXPERIENCES WITH ALGOL 68 TRANSPUT

PARTS I, II:

Critical remarks on Revised Report's transput section

by Hanno Wupper*  

July 1978

Preliminary remark

If, from the following pages, readers not familiar with Algol 68 might get the impression that part 10.3 of the Revised Report is a most useless and confusing document written by complete ignorants, the author must apologize and stress that this was not at all his intention. A useful and comfortable transput system has been defined there. The definition consists of a set of Algol 68 programs describing transput activities down to the handling of single characters in the file (or book, as it is called there), thus making clear beyond doubt what has to happen in any particular situation. Algol 68 transput, as well as its method of definition, is in a much better state than the definitions of I/O systems of various other languages and cannot be ignored by anyone working on operating systems or higher level languages.

Not before we actually started an implementation we noticed several disadvantages. The offensive tone of this paper is a result of disappointment due to expectations too optimistic. If some of the statements in this papers can be proved wrong, all the better. The author wishes to express his thanks to J.C. van Vliet from the Mathematical Centre, Amsterdam and to his colleagues G. Baszenski, J. Krieger, M. Peuser, N. Voelker, C.-G. Warlich and, above all, Prof. H. Ehlich, who all found time for long discussions of transput. Source of experiences was the work on an implementation for the TR 440, which would have been impossible without the fine Algol 68C bootstrap-kit from Cambridge University and J.C. van Vliet's machine independent transput system.

* Address: Rechenzentrum der Ruhr-Universitaet Bochum, Postfach 102148, D-4630 Bochum, W. Germany
Part I

UNDERSTANDING ALGOL 68 TRANSPUT

General considerations

1. Algol 68, defined by programs written in three different languages

In his remarks "On the Revised Algol 68 Report" [Algol Bulletin AB36.4.3] M. Sintzoff says that rewriting the old Report had been organized as a programming project. Indeed the Revised Report can be considered as a program defining the Algol 68 machine. It consists of three parts, each written in a different language:

- Syntax, i.e. the set of program texts accepted by the Algol 68 machine is defined by the rules of a two-level van Wijngaarden grammar. Such grammars in principle are powerful enough to simulate any Turing machine, thus allowing the occurrence of several indecidabilities; but in the Revised Report they have been used with care: Some "general hyper-rules" [RR 1.3] provide useful structured programming tools, and most of the syntax part certainly forms an elegant high level program with well chosen "identifiers" (i.e. paranotions).

- Semantics of the Algol 68 machine is defined by programs "expressed in natural language, but making use of some carefully and precisely defined terms and concepts" [RR 0.1.1] (cf. Sintzoff [AB36.4.3 p31]). The structure of the semantics parts is composed from serial, conditional, and case constructs and is closely related to the structure of the corresponding syntax rules.

- Only a nucleus of the Algol 68 machine, consisting of comparatively few "basic constructs" is defined by these means. The machine then is extended by a program written in Algol 68 itself: The important chapters on the standard prelude [RR 10.2] and transput [RR 10.3] mainly consist of operator- and procedure-declarations.

To implement Algol 68 on an arbitrary machine one should have to do nothing but provide translators for two-level grammars and for the special subset of natural language, and then simply "run" the Report.

Unfortunately this is a bit difficult.

Arbitrary two-level grammars cannot be automatically converted to a deterministic recognizer, and the special grammar in the Report seems not to be of a type for which parser constructors have been developed (cf. Deussen).

At least, however, it does not seem too difficult to hand-translate the rules to e.g. an equivalent affix grammar. (Affix grammars luckily may be automatically converted to an executable program, cf. Koster, Watt.)

Though as we see the definition of syntax and semantics as given in the Report is not suitable for automatic translation, it is clear and precise and tells the implementor exactly what to do - though, of course, "it may be difficult to understand to the 'uninitiated' reader".
The standard prelude as given in part IV of the Report [RR 10.22] in form of several Algol 68 routine texts serves as a useful implementation model: The method of definition here presents no difficulties; some of the operations might even be implemented by translating parts of the Report.

The transput section, however, [RR 10.3] presents a somewhat different situation. It again consists of Algol 68 declarations and again the method of description is understood easily; but then the implementor is left with long programs the intention of which tends to remain obscure. The additional pragmatic remarks sometimes are more confusing than enlightening. Moreover, some of the procedures even seem to contain bugs. In the following chapter we will have a closer look at why this section may be of little help to the implementor.

2. Initial problems: Finding out what to implement

When an implementor who has not been familiar with Algol 68 transput for years, studies section 10.3 of the Revised Report he finds himself completely lost.

At first he does not understand what books, channels, and files really are and what is the difference between them. He probably knows all about his own operating system and about the transput systems for several other programming languages; but the pragmatics of the Report give little help to match what he knows with what he is to implement for his Algol 68 system. Probably he sooner or later gets some idea that a "book" is more or less what is called file in most operating systems, and he surely will find out that a "file" is no more than a kind of status vector, describing the momentary situation of transput for an open book. But he gets quite mixed up when he wants to know what a "channel" really is. The pragmatics [RR 10.3.1.21] say something about physical devices, possibly useful in nuclear physics and that "a channel is a structured value whose fields are routines returning truth values which determine the available methods of access to a book linked via that channel" - which latter knowledge may as well be extracted from the definition of mode .CHANNEL, where it is expressed more clearly [RR 10.3.1.2.a]. One also learns that a channel has some "channel number", which slightly reminds the reader of Fortran's unit numbers. (The pragmatics say nothing about it, nor seems it to be used anywhere in the Report.)

A channel "corresponds to one or more physical devices" [RR 10.3.1.2] - but in modern operating systems the user does not need to know about physical devices. Maybe a channel really is a unit number plus some useful enquiry routines? But then one learns that several files may be open at one and the same channel a time. Perhaps it is safest to have just one channel and code everything in the file idf, or to provide one channel for each Fortran logical unit number and allow the idf to be the empty string only? The purpose of channels remains in the dark.

When after all one feels sure what to do with them one can start to implement a transput system. It will, of course, have to behave exactly as described by the Report. But it turns out to be more than a challenging puzzle to find out what really is described there. What one finally has found out often is rather astonishing and not what programmers might expect.

Of course this is a general programming problem. As soon as a language contains as powerful constructs as arithmetic operations, loops, and conditional expressions it may become indecidable what a program really does and whether it computes a certain given function.
Therefore, care should be taken first to specify the problem, then to write a well structured program containing assertions to prove its correctness. Most programs are written the other way round: The programmer's vague idea of what the code should look like is punched in; during the "testing phase" several conditional statements are added, and finally the problem is adjusted to the behaviour of the program.

Several important languages are known to be defined as "what the compiler accepts", and some unexpected unorthogonal restrictions in the Fortran standard may have resulted from the same technique.

Sadly enough, the transput section of the Revised Report gives a similar impression - it even seems to contain serious bugs. Several presumed errors and other problems are listed in an interesting paper compiled by van Vliet [DWA 11/13]. We will mention only a few of them:

- The .BEYOND operator used in establish [10.3.1.4.b] does not test whether a given position is "beyond" another one.
- Even though a special "primal envirom" has been introduced some transput calls will violate scope restrictions.
- By definition of mode .INTYPE [10.3.2.2.d] it is not possible to input values of e.g. mode .STRUCT(.BOOL b, .STRING s).
- Input and output are incompatible in several cases.

Hopefully, the programming errors and misprints will be corrected by some official document; more dangerous are the cases where the behaviour of the routines is obscure, or unexpected, or not realistic: Implementors will be likely to deviate slightly or less slightly from the Report. There are already several implementations of different transput systems, not all of them being super- or sublanguages.

But even if one wants to stick to the Report absolutely and implement the transput routines exactly as they are printed there one is in a mess:

Most parts of transput activities are performed by the operating system. What remains to be done by language dependent routines, besides conversion of values, is the testing of conditions (or "events"), calling event routines, providing default actions, etc. In Algol 68 these actions are strongly connected with routines provided by the user. They are quite complicated logically, but not critical in the sense of CPU time, so it would be sensible to copy them from the Report as they are. However, it turns out to be impossible to draw a line dividing chapter 10.3 of the Report into two appropriate parts:

Especially the layout routines are a conglomerate of actions belonging to all the different layers in an operating system, from user level right down to the physical device.

One cannot copy the routines from the Report; so one has to write a completely new transput system; so one is forced to understand what is defined by the Report, or, more precisely, what is intended there.

Studying the transput section and finding out its intention turns out to be a source of not "innocent merriment" but innumerable surprises.

Possibly the authors had in mind a special, limited machine or operating system with some very particular restrictions. The decision to use the same set of layout routines for both reading and writing causes additional restrictions and problems.
PART II

ALGOL 68 TRANSPUT CONSIDERED INSUFFICIENT

The following chapters try to show that even if all implementation problems are solved one cannot feel too happy because the result will be disappointing.

While implementing Algol 68 transput on our TR440 we came along a number of problems far more serious than the above mentioned difficulties in connection with the method of description: transput, implemented exactly as it is defined in the Report contains a number of unorthogonal restrictions that will surprise the user and possibly give him the impression that Algol 68 possesses one of the most old-fashioned transput systems he finds on his machine. Handling of random access will be inefficient and undesirable, even in a "superlanguage" where some of the unnecessary restrictions have been dropped.

4. Superfluous restrictions

The attributes set_possible, get_possible, put_possible, and bin_possible ought to be mutually independent. There is no reason why changes between char_mood and bin_mood should be forbidden whenever NOT set_possible, or why reading and writing may not be alternated in the singular case of a sequential file used for binary transput.

5. Important types of books not considered at all

The texts of books for Algol 68 always have to resemble a .REF .FLEX [] .FLEX [] .FLEX [] .CHAR; each character being identified by a triple of integers, the page, line, and character numbers. Positions inside the logical file run from .POS(1,1,1) and never contain gaps.

Modern operating systems provide books of a somewhat different structure (sometimes called "index sequential" or "sequential keyed"): There are "record keys" rather than line numbers, taken from an ordered set of, in general, strings or sometimes integers. Arbitrary gaps between keys inside the logical file are possible. Such books may be read sequentially or set to a certain line or to the first existing line after a certain key. Lines may be of arbitrary length. Usually there are no pages, but the concept may be generalized by allowing arbitrary tree structure rather than adding just one more dimension for pages.

Books of that kind or at least useful special cases can be handled by several programming languages and operating systems. The special case with integral keys is quite common, even to certain Basic implementations. Due to its limited nature, standard Fortran I/O can be extended to allow handling of such books in a straightforward way. The more complicated Algol 68 transput, however, suggests more than one way to extend the effects of newline, space, or set, and certainly there are several possibilities for additional layout routines and events (on undefined record, delete). Some official document with appropriate recommendations is needed urgently, otherwise there soon will be all kinds of differing implementations.
Another type of book not provided for is the good old magnetic tape: Books on tape may be read backwards! (In most operating systems that is extended to all sequential files on background storage. In Algol 68 the routines space/backspace are not really symmetric and there are no reverse versions of newline or newpage. The effect of Fortran's BACKSPACE can be achieved by no means. Last not least there are no hints how to make use of a dialogue terminal.

6. Random access condemned to inefficiency

The main difference between sequential and random access is that in the latter case the logical file end always is close to the character written last, while with random access lines may be overwritten without destroying the information on subsequent lines. The Report tries to make users benefit from the advantages of random access as much as possible - with the result that after each call of newline or set, the old contents of the new line must be available. There are no means to express that one wants to overwrite an entire line without reading it first, though that is the case in most applications.

If the text of a book actually is a multiple value accessible to the program this is, of course, no problem. In a real life operating system, however, where records have to be read from background storage ("or even from a set up in nuclear physics"), it can be too expensive. Perhaps pairs of routines like set_to(p,l,c) vs. get from(p,l,c) might be what is needed (with space writing blanks if a line was positioned by set_to).

7. Unrealistic behaviour of compressible random access books

Except for the restrictions and inefficiencies mentioned above, all incompressible and all sequential books behave sensibly. Now in [RR10.3.1.6.aa] a most interesting pragmatic remark can be found:

"Although the effect of a channel whose books are both compressible and of random access is well defined, it is not anticipated that such a combination is likely to occur in actual implementations."

At first glance no harm seems to be in that statement, though, of course, assumptions on what manufacturers might provide in an operating system could be hazardous. Compressibility is orthogonal to and independent of random accessibility. Users of our TR440 in fact have been declaring nearly all their books compressible with random access for the last eight years. Both properties are implemented efficiently, and when the price is low one usually chooses the best available.

The shock occurred when, after starting an implementation of Algol 68 transput, we had a closer look at what actually is well defined there: Once a line has been written to a random access book its length cannot be changed any more (except by scratch), even if the book is compressible. Just look at the last example of [RR 10.3.1.3]: It will not work if set_possible (f1) .AND. compressible(f1) and if the lines of the book happen to be shorter than int width.

This is a severe restriction and quite against the philosophy of random access.
CONCLUSION

If Algol 68 is meant to be a serious alternative to other existing languages, a revision and extension of transput is needed.

It should be stated here that C.H. Lindsey's recent article "Algol 68 and your friendly Neighbourhood Operating System" [AB 42.4.4 p.22] probably will be of great help to all implementors. Lindsey gives a comprehensive overview of implementation problems and possible solutions. Indeed many of the problems mentioned occurred to us during the last year. We found similar solutions, but not without long discussions and a lot of thinking. Most of the problems we felt unable to solve, however, are not dealt with by Lindsey.

Literature


