THM

ABTEILUNG MATHEMATIK

TECHNISCHE HOCHSCHULE MÜNCHEN, 8000 MÜNCHEN 2, ARCISSTRASSE 21

ALGOL 68 M

by

U. Hill, H. Scheidig, H. Wössner

Technical University of Munich

Report Nr. 7009

The "ALGOL 68 group" of the Technical University of Munich has been concerned for the past two years with the implementation of ALGOL 68. Essentially, the full language is handled, without the facilities of parallel processing and the synchronization operations, and with some further "technical" restrictions. The implementation is nearly completed, and we expect that the compiler will go into multi-test-phase during 1971. The following pages attempt to summarize our experiences resulting from this intensive effort with ALGOL 68.

First of all, it should be noted that ALGOL 68, due to the extent and content of the language and the consistency of its description, is extremely well suited for studying and using nearly all the general concepts and principles which have been included in any of the present day higher level programming languages.

In consequence of this high generality of ALGOL 68, however, the size of an ALGOL 68 compiler will reach a multiple of that of an ALGOL 60 compiler (in our case, as compared to the existing ALGOL 60 compiler, the factor will be about 6). For the translation time there will be a similar increasing factor, and also for "normal programs", that is, for relatively simple programs which do not use extreme properties of the language. The great latitude in the representation of the ALGOL 68 symbols (especially, the overloading of the round parentheses and the syntactic ambiguities occurring in connection with the application of indicants) and those features of the language which are rather rarely used, force the compiler to an organizational expense which also burdens the translation process of normal programs.

Now, one can argue that the increase of translation time is not so important as long as the run-time efficiency of the program is guaranteed. To this point of view we must say that, on the one hand, the increasing factor of compiler size and translation time, is too great to be completely neglected and, on the other hand, that certain problems arise which affect as well the run-time efficiency of an ALGOL 68 program. For example, the handling of local generators and intermediate results (particularly multiple values), and the storage allocation for certain non-elementary values introduce some problems which lengthen the object program as well as its execution time. If we must, as in the last example, reserve storage in the working stack for a non-elementary value - the mode of which can be defined by means of indications, especially when recursive - we are concerned with two tasks: We have firstly to elaborate the constituent boundscripts, if any, and to make the necessary checks and, secondly, to determine the storage needed by the considered object. These processes have to look into the mode of this object, what can be performed by generating all corresponding instructions in the object program or, "half-interpretatively", by referring to an entry in a table which contains representations of all modes used in the program.

So, it turns out for ALGOL 68 that, compared to ALGOL 60, the compiler length and the translation time will be considerably increased and that we must also expect a certain loss of efficiency at run-time. The question is now whether this is a necessary price to pay for having a better and more comfortable language. To this point, we admit that we must of course, pay for such a language. But the expense can be considerably reduced by imposing certain restrictions to ALGOL 68 which are to a large extent of a notational nature and which do not reduce the power of the language very much. Such restrictions would

especially take note of the "normal user" whose predominant interest will be that the language enable him to formulate his problems without requiring excessive time for learning the language. We should take into account that many of the people here denoted as "normal users" already have difficulties in understanding some concepts of ALGOL 60 (e.g. procedure and block concept). It will be very hard for such people to get accustomed to the use of the full ALGOL 68; in particular, it will be difficult for them to come to an adequate understanding of the whole system of coercions and connected facilities.

The restrictions for ALGOL 68 which we consider to be desirable from the implementors point of view as well as from the aspect of teaching and learning the language are listed in the following sections. (By changing a feature of the language, the obvious consequences for involved extensions and for the wording of the semantics are not explained.) ALGOL 68 restricted and changed in the following way is called "ALGOL 68 M".

I. Substantial changes

1. Modes and declarations

1.1. Restrictions for modes

The following changes concerning the metaproduction rules of MODE are introduced:

1.2.1. c) TYPE: PLAIN; format; PROCEDURE; REFSETY NONREF.

1.2.1. h) LONGSETY: long; EMPTY.

1.2.1. o) STOWED: structured with FIELDS; ROWS NONSTOWED; ROWS REF STOWED.

1.2.1. q) FIELD: REF ROWS NONSTOWED; REFSETY structured with FIELDS; REFSETY NONREST.

The following metaproduction rules are added:

1.2.1. z) REF: reference to; reference to reference to.

1.2.1.aa) REFSETY: REF; EMPTY.

1.2.1.ab) NONREST: UNITED; PLAIN; format; PROCEDURE.

1.2.1.ac) NONREF: NONREST; STOWED.

This restricts the number of long's and ref's to 1 and 2, respectively; further, an element of a multiple value (structured value) cannot be a structured value (multiple value). To extend these restrictions also to union-modes, the corresponding productions in section 7.1.1 must be changed in an obvious way.

By this restriction on modes, on the one hand, the problem pointed out above concerning the storage allocation and checking process (at run-time) for values of non-elementary modes becomes easier and, on the other hand, copying operations occurring in connection with assignations of such values are minimized.

For example, the mode-declaration

struct m = (..., [1:n] **real** x, ...)

is no longer allowed whereas

struct m = (..., **ref**[:] **real** x, ...)

still is.

1.2. Reintroducing the void symbol

The representation of the void symbol is reintroduced (as proposed in Habay-la-Neuve) in

7.1.1.z) virtual void declarer: void symbol.

In 8.3.0.1, correspondingly, MODE is replaced by MOID.

1.3. The declaration condition

Section 4.4.4 is extended by the following condition:

4.4.4.d) If the actual-declarer of a mode-declaration is or contains a mode-indication, then the occurrence of this indication is called a "neutral" occurrence.

4.4.4.e) No proper program contains an indication-applied occurrence which is not neutral and which is not preceded by the indication-defining occurrence identified by it.

Therefore, the following is no proper program:

```
begin s s; struct s = (...); ... end
```

but

```
begin struct s = (int a, t b);
      mode t = real;
      s s; ...
end
```

is a proper program (with respect to this new condition).

With this restriction, the compiler can, during the syntactical analysis, easily distinguish between mode-indications and adic-indications, and all difficulties which otherwise can occur in connection with indications are removed.

2. Constructions

2.1. Collateral clauses

In 6.2.1 the following changes are introduced:

6. 2. 1. c) STRONGETY collateral row of MODE clause: row symbol, STRONGETY MODE unit list proper pack.

6. 2. 1. d) firm collateral row of MODE clause: row symbol, firm MODE balance pack.

At all other places in 6. 2. 1, "PACK" is to be replaced by "pack". By this change a clear distinction between collateral row of MODE clauses and structured with FIELDS clauses is made. Therefore, the balancing process for collateral clauses is simplified and the identification of operators becomes easier; in particular, for the uniqueness condition 4. 4. 2 the version of MR 99 is sufficient.

2. 2. Generators

Local generators are dropped out everywhere with exception of the right hand side of identity declarations by the following replacements:

7. 4. 1. a) identity declaration: formal MODE parameter, equals symbol, general actual MODE parameter.

7. 4. 1. c) general actual MODE parameter: strong actual MODE parameter; MODE local generator; special MODE assignation.

8. 3. 1. 1. d) special reference to MODE assignation: special reference to MODE destination, becomes symbol, MODE source.

8. 3. 1. 1. e) special MODE destination: MODE local generator.

8. 5. 1. 1. a) MODE generator: MODE global generator.

The use of global generators is further restricted by excluding them as boundscripts.

The meaning of the last restriction is obvious; for the first one see [2]. That paper outlines that we can handle local generators by introducing an order for the elaboration of assignations and row and structured displays. We quote here the last sentence (of the conclusion) of [2]:

"However, one can ask oneself whether it is sound to determine an order in a collateral elaboration for this purpose and not for e.g. optimization of code generation. The other alternatives are either to complicate the organization of the working stack, which results in run-time inefficiencies, or to reserve locations on the heap which is from a conceptual point of view not very attractive."

These and other considerations lead us to remove local generators as cohesions (for local generators the same problem can arise as mentioned in II). [1]

2.3. Reintroducing the depression

In certain situations it can be useful to have an explicit notation for a dereferencing operation. Therefore, the MODE depression is reintroduced in the same way as in MR 99.

2.4. Call

We replace 5.4.1.c and d by:

5.4.1.c) VICTAL PARAMETERS and PARAMETER: VICTAL PARAMETERS, comma symbol, VICTAL PARAMETER.

The effect of this restriction is that the identity declarations resulting from the formal and actual parameters pack are not serially elaborated. In fact, their elaboration is defined as "quasi-collateral" in the following sense: At first, all actual parameters are elaborated in turn and the actual values are stored in successive cells of the storage part reserved for the calling routine. Subsequently, the routine is entered, which means that the organization required for procedure calls is performed, e.g., storing of organizational data, loading of an index register for adressing

---

1) This point and another one following in 2.4 was discussed with the Brussels M.B.L.E. ALGOL 68 group.

purposes. This concept avoids unnecessary recursive calls caused by the actual parameters. Now, before elaborating the routine, the declarers of the formal parameters pack are evaluated if they contain any expressions, and checks of formal and actual boundlists are made.

The main reason for introducing the serial elaboration of parameters was to allow side effects as in the example

```
proc p = ( [1 :] real a; [1 : upb a] real b).
```

This effect ist preserved by the quasi-collateral elaboration of

```
proc p = ( [1 :] real a, [1 : upb a] real b).
```

But, it is no longer possible to have any defined side effects between formal and actual parameters, as intended, e.g., by

```
proc p = ( [1 : (n := 3)] real a; int b) : ··· ;
··· p ((1, 2, 3); n).
```

Reasons for this restriction are:
One of the aims in designing ALGOL 68 was to improve and simplify the procedure concept of ALGOL 60, especially by eliminating the "name-calls", which require the actual parameters to be transformed into subroutines. But, in fact, these name-calls are still necessary, at least in the phase of the actual parameters transfer. More precisely, if the i-th and the (i + 1)-th formal parameter are separated by a go-on-symbol the formal declarer of the i-th parameter must be evaluated before the (i + 1)-th actual parameter. This means in practice, that the compiler has to transform either actual parameters into subroutines which may be called by the program parts produced from formal parameters, or formal parameters into subroutines which may be called by the object program parts corresponding to the actual parameters. The consequences are, in principle, the same for both possibilities. For both practical and aesthetic reasons we prefer the first solution which corresponds to the methods generally used for

ALGOL 60. (Of course, once the parameters transfer is completed, the further handling of the parameters is simpler than in the case of ALGOL 60, since new subroutine calls are not needed.)

The compiler, then, has to preserve the following tasks: the routine is entered before evaluating actual parameters, the formal parameters are evaluated and the actual subroutine corresponding to each is immediately called. Such a subroutine call requires a similar effort as a procedure call, with all consequences, concerning reloading of index registers, e.g.

The ALGOL 68 Report does not define any correspondence between the sequences of gommas in the formal or virtual and the actual parameters pack. Therefore, the compiler has no information on whether the parameters are to be elaborated collaterally or not, when handling procedure calls (especially not, if the primary of the call is an expression or a formal parameter).

The non-collateral case therefore must always be preserved. Nevertheless, there are certain optimization possibilities: (1) all parameters can be evaluated collaterally (without using subroutines), if no bound check is needed; (2) if the i-th parameter, $i = j, \ldots, k-1$, does not contain boundlists, but the k-th does, then these parameters can all be handled collaterally, that is, eventually, transformed into one single subroutine. These optimizations can be easily implemented, since no prepass or special search mechanism is required.

But, in spite of such optimizations, compiler as well as object program and run-time remain burdened and, furthermore, since this concept will in general not be used by "normal users" who avoid, as it is well known, constructions whose consequences are not immediately obvious, the restriction to quasi-collateral handling of parameters is made.

2. 5. Repetitive statement

The repetitive statements are changed in the following way:

a) "do E" is, at all places in 9. 3, replaced by "do E done".

b) If E is, in particular, a strong-closed-void-clause and S its serial clause, then "do E done" may further be replaced by "do S done".

Finally, it could be useful to introduce a facility similar to the for-list (with more than one element) in ALGOL 60.

3. Coercions

3. 1. Proceduring and rowing

The coercions proceduring and rowing are dropped out (by removing the sections 8. 2. 3 and 8. 2. 6).

In consequence, the following changes are made:

5. 4. 1. a)* routine denotation: PROCEDURE denotation.

5. 4 1. b) procedure with PARAMETERS MOID denotation: routine symbol, formal PARAMETERS pack, MOID cast.

The following rule is added:

5. 4 1. g) procedure MOID denotation: routine symbol, MOID cast.

Correspondingly, the void cast pack is removed from 8. 6. 0. 1. b (see I. 1. 2.).

3. 2. Extension of the selection

In connection with the documentation of our ALGOL 68 Compiler we had the experience that, if we use ALGOL 68 itself as description language, it is very awkward in that we have, by means of a selection, no direct access to an object the mode of which is a union mode and the value is a structured value. In this case we always have to go through a conformity

operation even if we are sure about the actual value. This fact makes the description of the compiler (in ALGOL 68) unnecessarily complicated and long.

Therefore, we change 1. 2. 3. m and make the following additions:

4. 2. 3. m) FITTED: dereferenced; deprocedured; decomposed.

1. 2. 3. n) WEAV: weak; exclusive.

1. 2. 3. o) STAIR: STIRM; fair.

8. 2. 0. 1. b) exclusive COERCEND: COERCEND; exclusively FITTED to COERCEND.

8. 2. 2. 1. d) exclusively deprocedured to MODE FORM: procedure MODE FORM; fairly FITTED to procedure MODE FORM.

In 8. 2. 1. 1.a and 8. 2. 2. 1.a "STIRMly" is replaced by "STAIRly" and in 8. 2. 1. 1.b "weakly" by "WEAVly".

A new section 8. 2. 9 is added:

"8. 2. 9. Decomposed coercends

{Coercends are decomposed when it is required that the a priori mode should be changed from 'union of LMOODSETY structured with FIELDS RMOODSETY' in 'structured with FIELDS', e. g., in a of x when x is declared as union (int, struct (int a, real b)) x.}

8. 2. 9. 1. Syntax

a) exclusively decomposed to LREFSETY REFSETY structured with FIELDS FORM: fairly FITTED to LREFSETY union of LMOODSETY REFSETY structured with FIELDS RMOODSETY mode FORM; LREFSETY union of LMOODSETY REFSETY structured with FIELDS RMOODSETY mode FORM.

b) fairly decomposed to REFSETY REPROTY structured with FIELDS FORM: fairly FITTED to REFSETY union of LMOODSETY REPROTY structured with FIELDS RMOODSETY mode FORM; REFSETY union of LMOODSETY REPROTY structured with FIELDS RMOODSETY mode FORM.

8. 2. 9. 2. Semantics

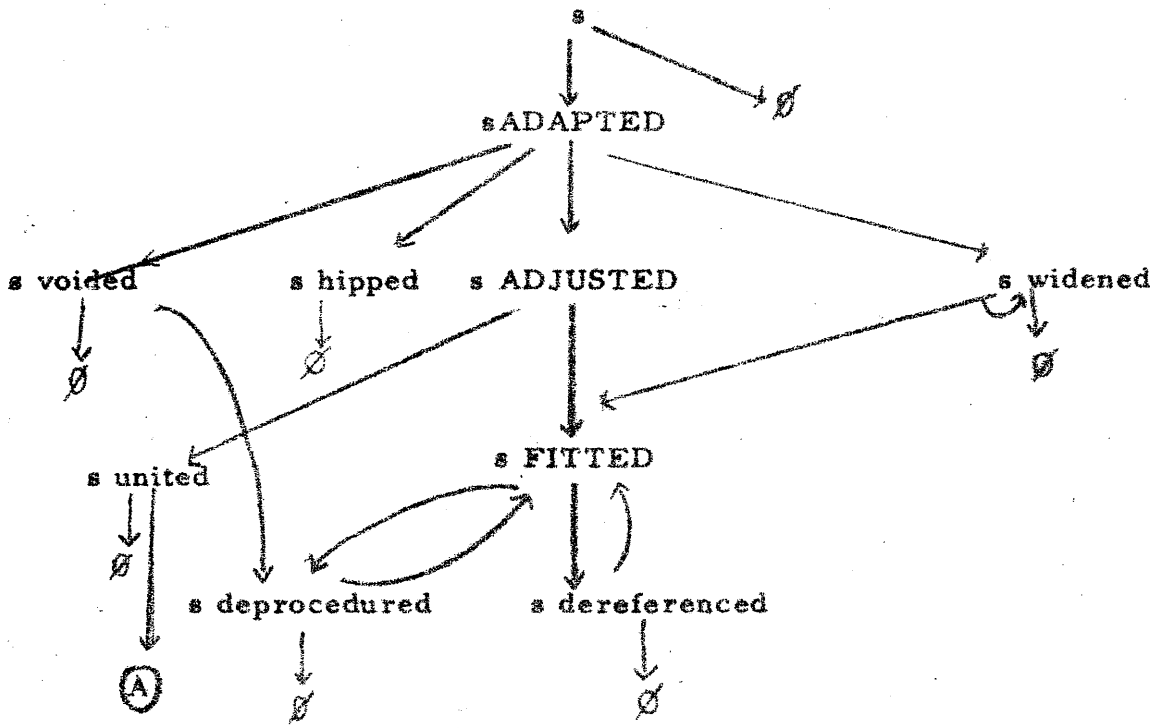
A decomposed-coercend is elaborated in the following steps:

Step 1: It is preelaborated; the value yielded is considered.

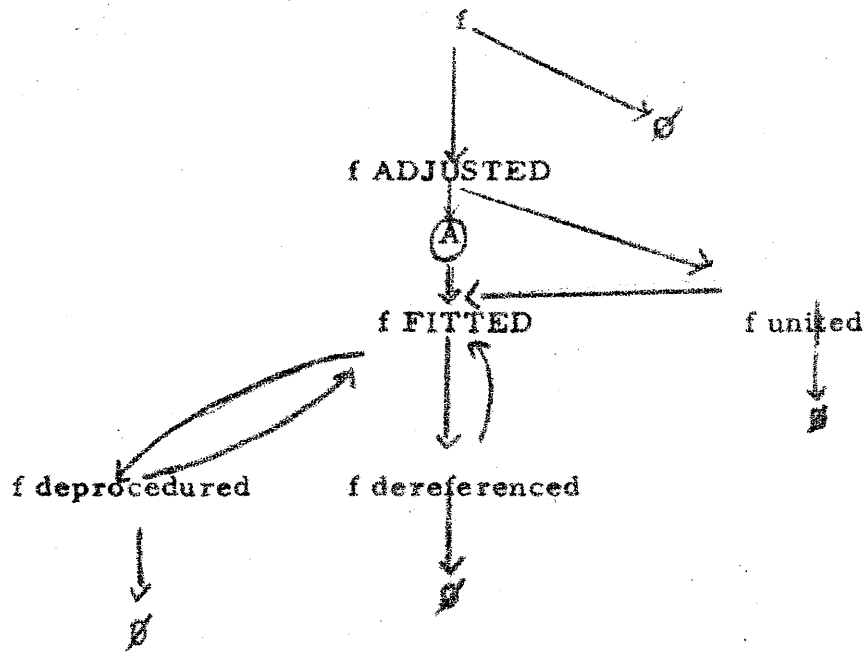
Step 2: If the mode of the considered value is enveloped by the original of the decomposed-coercend, then the considered value is the value of the decomposed-coercend. Otherwise the further elaboration is undefined."

We give now a schematic description of the coercions after these changes by means of 4 graphs. The modes of these graphs correspond to the modified productions from [1] 8. 2 and the direction of the arrows corresponds to the direction of the productions.

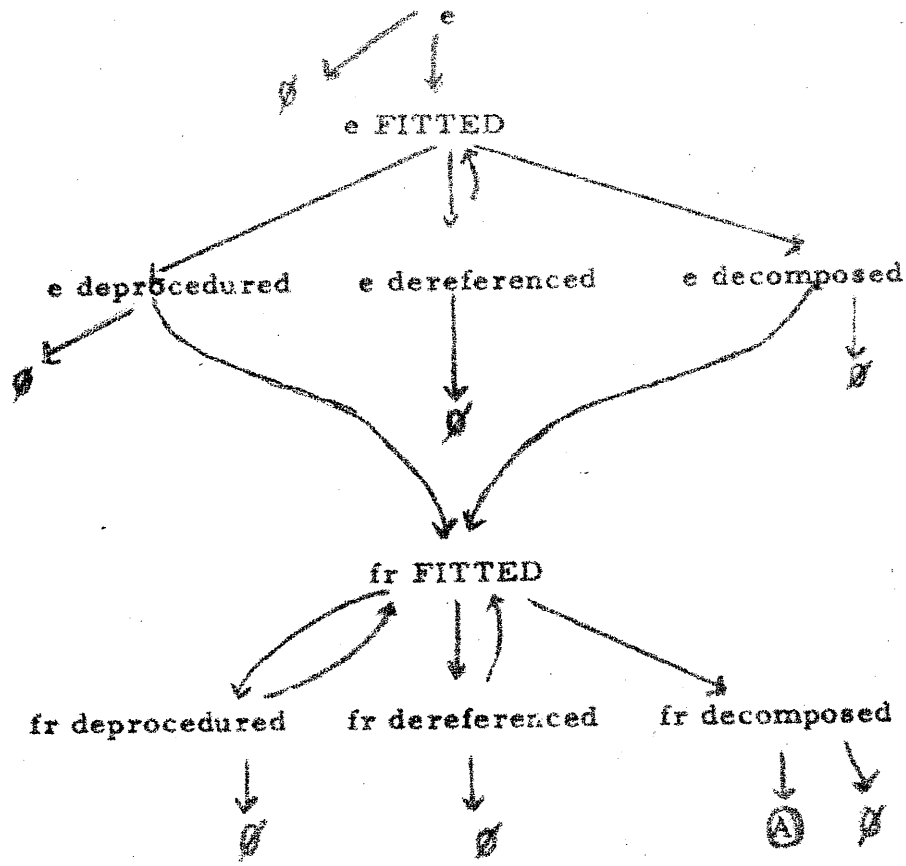
a) strong coercions (s):



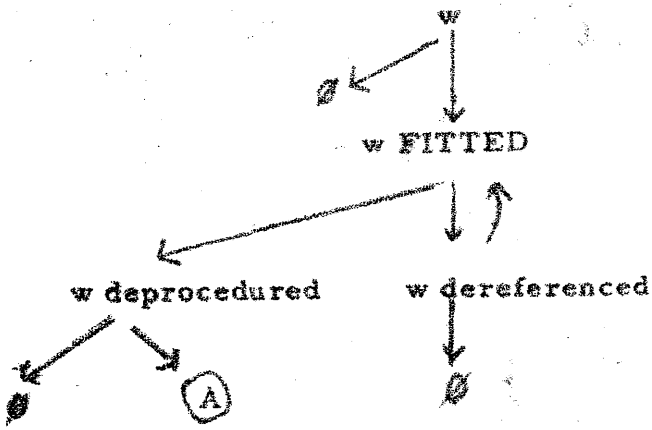
b) firm coercions (f):



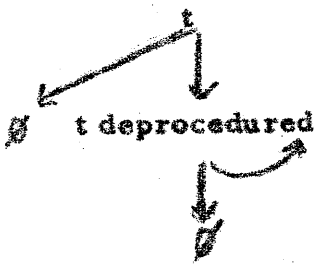
c) exclusive (e) and fair (fr) coercions:



d) weak coercions (w):



e) soft coercions (t):



Now, 8. 5. 2. 1. a is replaced by

8. 5. 2. 1. a) REFETY MODE selection: MODE field TAG selector,
of symbol, exclusive REFETY structured with
LFIELDSETY MODE field TAG RFIELDSETY
secondary.

Example: begin
struct s1 = (int a, real b);
struct s2 = (int a, bool c);
union u = (s 1, s 2);
u u : = if condition then (1, 1. 0) else (0, false) fi;
a of u
end

Depending on the condition the value of the foregoing closed-clause is 1 or 0.

If we want to avoid this high degree of generality which allows u to be of a mode united from two structured modes both containing the field identifier a, then we may add the following restriction:
If the secondary of a selection S is a decomposed coercend C then the mode enveloped by the original of the (only) direct descendent of C may not be united from (at least) two structured modes which both contain a field identifier identical with the selector of S.

The check of this condition can be easily performed by the compiler, so that - at least with this restriction - the given generalization of the selection does not essentially burden the implementation.

II. A further restriction concerning slices

Let us consider the following example:

begin
int a;
a:= begin [1 : 5] int b:= row (1, 2, 3, 4, 5); b end [tert]
end

At run-time, after the elaboration of the inner closed-clause the working stack for this closed-clause contains the multiple value which is its result. If we leave the inner closed-clause then, of course, we want to give up the space on the working stack belonging to it, but in that case this is not possible because the "intermediate" result is still needed. Now, we have two possibilities (applying the usual stack technique):

a) We do not give up in such cases the storage containing the result.

b) We transport the result into this part of the working stack which belongs to the embracing closed-clause.

The last method is very time-consuming because, in general, we cannot make this transport "en bloc"; but, if we want to avoid this transport of intermediate results - by the first method - we loose storage on the working stack (of course, the same can occur in connection with procedure results).

So, if we want to remove this problem - this can be important if the storage of the available computer is small - we introduce the following restriction concerning the use of slices.

8. 6. 1. 2. Step 2 is changed by:

"Step 2: The multiple value which is, or is referred to by, the value of the primary, is considered; if the considered value is inadmissible {by the following definition}, then the further elaboration is undefined; otherwise, a copy ...".

Definition of "inadmissible":

The value V of a block (see III) or a routine denotation C is called "inadmissible", if it is the value possessed by either

a) a collateral row of MODE clause,

b) an applied occurrence of a row of MODE identifier identifying a defining occurrence contained in C, or

c) a row of MODE slice.

III. Notational changes

The changes are listed in the following table:

Section in [1]	symbol	representation		remark
3.1.1.b	formatter -	f	$	removed
	flip -		1	"
	flop -		0	"
	space -	blank	.	"
	binary	b		new
	octal -	o 0		"
	hex -	h		"
	digit ten -	a		"
	digit eleven -	b		"
	digit twelve -	c		"
	digit thirteen -	d		"
	digit fourteen -	e		"
	digit fifteen -	f		"
3.1.1.c	over and becomes -	overb	÷:=	removed
	modulo and becomes -	modb	+::=	"
	and -	and	∧ &	"
	is at most	<= le	≤	"
	is at least	>= ge	≥	"
	over -	over	÷	"
	modulo -	mod	÷:	"
	th element of -	elem el	□	"
	power -	↑ up		new
	lower bound of -	lwb	⌊	removed

Section in [1]	symbol	representation	remark
3.1.1.c	upper bound of	upb	⌈ removed
	lower state of -	lws	⌊ "
	upper state of -	ups	⌈ "
	plus i times -	i	⊥ ! "
	not -	¬ not	~ "
	down -	down	↓ "
	up -	up	↑ ^ ** "
	becomes -	:= :=	.. = "
	conforms to and becomes -	::= :.= ctab	:. = new
	is not -	≠: is not .≠:	isnt removed
3.1.1.d	void -	void	new
	priority -	priority prio	prio "
3.1.1.e	sub -	[(/	(/ "
	bus -	] /)	/) "
	at -	at	@ removed
	if -	if	(case "
	fi -	fi	)esac "
	case -	case	new
	in -	in |	"
	out -	out (	"
	esac -	esac	"
	of -	of	→ removed
	routine -	routine expr	new
	row -	row	"
3.1.1.g	skip -	skip	~ removed
	nil -	nil	° "
3.1.1.i	quote -	"	quote "
	comment -	co comment	¢ # "

Of course, such changes are made with regard to a special hardware. So, the representations

÷ ∧ ≤ ≥ □ ⌊ ⌈ ⌊ ⌈ ⊥ ↓ ^ ↑ → ∘ ~ ∼ ¬

are not available in the character set of the I/O units of or our installation (card reader and printer using IBM 026H code) and, therefore, are removed.

According to the new representations for sub- and bus-symbol, the divided-by-symbol is, in addition to the equals- and the times-symbol, only allowed as dyadic operator. (The corresponding changes of 3.0.4.a, b and 4.2.1.c are obvious.)

The changes in 3.1.1.e allow a simple analysis of closed-, conditional- and case-clauses in an obvious way. In order to avoid the remaining overloading of the open- and close-symbols, we remove the extension 9.2.g and, moreover, adopt the following changes:

6.3.1.a) SORTETY closed MOID clause: SORTETY MOID block package; SORTETY MOID compound pack.

6.1.1.a) SORTETY serial MOID clause: SORTETY MOID block; SORTETY MOID compound.

6.1.1.m) SORTETY MOID block: declaration prelude sequence, SORTETY MOID compound.

6.1.1.n) SORTETY MOID compound:suite of SORTETY MOID clause trains.

6.1.1.o)* serial clause: SORTETY serial CLAUSE; SORTETY MOID block; SORTETY MOID compound.

Finally, we introduce the following change of the bits-denotation (see[3]), replacing 5.2.1.c and 3.0.3.e by

5.2.1.c) structured with row of boolean field letter aleph denotation:
binary symbol, zeroone sequence;
octal symbol, octal sequence;
hex symbol, hex sequence.

5. 2. 1. d) zeroone: digit zero; digit one.

5. 2. 1. e) octal: digit zero; digit one; digit two; digit three; digit four; digit five; digit six; digit seven; digit eight.

5. 2. 1. f) hex: DIGIT; digit ten symbol; digit eleven symbol; digit twelve symbol; digit thirteen symbol; digit fourteen symbol, digit fifteen symbol.

In 5. 2. 2 all occurrences of flipflop, flip-symbol, and flop-symbol are replaced by zeroone, digit zero, and digit one, respectively; moreover, at the beginning of section 5. 2. 2 the following sentence is added:

"If the bits-denotation starts with an octal-symbol then the octal-symbol is replaced by a binary-symbol and each octal is replaced by three zeroones which represent the octal in the binary number system.

If the bits-denotation starts with a hex-symbol then the hex-symbol is replaced by a binary-symbol and each hex is replaced by four zeroones which represent the hex in the binary number system.

After these replacements, if any, the value of the bits-denotation is obtained as follows:"

References

[1] A. van Wijngaarden (Ed.), B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster, Report on the algorithmic language ALGOL 68. Num. Math. 14(1969), 79-218.

[2] P. Branquart, J. Lewi, and J. P. Cardinael, Local Generators and the ALGOL 68 Working Stack, Technical Note N62, M. B. L. E. Brussels, September 1970.

[3] G. Goos, Eine Implementierung von ALGOL 68. Report Nr. 6906, Rechenzentrum der Technischen Universität München, 1969.