June, 1968

Gier Algol 4

S T A C K A N A L Y S I S

by

Boris Martynenko

Acknowledgments

This paper is a result of the author's work at A/S Regnecentralen during September, 1967 - June, 1968 in accordance with the agreement on exchange of trainees between the Ministries of Higher education of the Kingdom of Denmark and the USSR.

The subject was suggested by Peter Naur, the scientific adviser of the work.

All the machine code pieces in the program were improvised by Jørn Jensen. He and Søren Lauesen were the main suppliers of the working information.

Jørgen Zachariassen essentially facilitated the debugging of the program.

The manuscript was corrected by Donald Wagner.

Their help and the support of many others, who made the work pleasant, are greatfully acknowledged.

Of course, the entire project would have been impossible without the kind reception of the director of the firm, Mr. Niels Ivar Bech.

June, 1968
Gier Algol 4

PROGRAM FOR STACK ANALYSING

Introduction

Program ANALYSIS is designed for using after an emergency situation has happened during running a program translated with the help of the GIER Algol 4 compiler. (The cases of the execution termination with alarm messages are listed in A MANUAL of GIER ALGOL 4, Appendix 2,pp. 78-79).

It prints out (by-value is equal to 17) the contents of the program stack at the moment of the emergency. This is in a form, suitable for the user and includes maximum possible information for his advantage to identify the stack items with the objects of the program.

The only thing ANALYSIS needs is the core store image dumped on the drum.

It can be applied in two modes depending on the state of the KB-register manually controlled:

1. The values of the array elements are printed, if the KB-register is on.

2. The so called array pictures are printed, if the KB-register is off.

The array pictures supply some squeezed information about the values of their elements.

Any number of shifts between the two modes are allowed at any moments during running ANALYSIS.

June, 1968
Gier Algol 4

PROGRAM FOR STACK ANALYSIS

Instructions to the operator

The binary paper tape BINARY ANALYSIS looks like the following:

```
 move, image, free<
 res, stack<
 binin, free<
[BINARY ANALYSIS]
 res, analysis<
 run, analysis<
 clear, analysis<
 clear, stack<
 t<
```

Thus the normal order for using it is as follows:

1. Upon receiving an alarm message during the run of a translated program - dump the core store image unless it has been done by the system.
2. If the permonent compiler has been used, the command
   ```
   clear, ga4<
   ```
   should be typed.
3. Load paper tape MOVE.
4. Load BINARY ANALYSIS by inserting the paper tape into the reader and typing: r<

Execution of the program terminates with the message: 'analysis' and the usual interruption message with the typewriter selected as Help's current input medium (see A MANUAL OF HELP 3, p.13).

After this it is possible to repeat the run of ANALYSIS without its reloading by typing the command:

```
run, analysis<
```

or to remove it, together with the area 'stack', which contains a copy of the core store image, from the drum by typing the command:

```
r<
```

June, 1968
Gier Algol 4

Structure of the stack picture

The contents of the stack as represented by program ANALYSIS consist of:

1. An emergency stack reference indication:
emergency sr = < sr >,
< sr > being the current stack reference at the moment of the emergency.

2. Heading: s t a c k
followed by the contents of the stack (3-10).

3. Heading: o w n
followed by the values of all the own variables of the program, if such are declared. Within each block the own variables appear in the order in which they are declared. Like other simple variables they may appear in 2 to 4 alternative formats (see 7. Heading: variables).
The blocks are given in reverse order.

4. Heading: < sr >: block in < sr0 >
or < sr >: inactive block in < sr0 >
indicates the begin of a block or a procedure body with < sr > as its stack reference.
< sr0 > is the stack reference of the surrounding block. In the case of the procedure body, this is the block where the procedure is declared.
For the outermost block < sr0 > = 1.
The second form of the heading corresponds to a block which is reserved in the stack but whose stack reference is not in the display.
For example, an actual parameter may be a function designator with the identifier declared in the same block as the procedure called; or a procedure call may include as one of its actuals another function designator with the same identifier (even if it corresponds to a parameter called by value - no recursion); or the block may be that of a procedure. The emergency has happened during the evaluation of the actual parameters or during one of the several incarnations of the procedure body.

Particularities:

4.1. If a program is not a block but a compound statement, it is considered to be the outermost block.

4.2. A procedure body is always considered to be a block, even if it is formally not such.

4.3. Some SLOW standard procedures such as

where, cancel and reserve

create a block (of the first level) in the stack. The treatment of the contents of it is described in GIER ALGOL 4 LIBRARY PROCEDURES, 1967.

The heading mentioned above is followed by the items which are actually in the block. They are (5-8):

5. The value of the function, if the block is a procedure-function body:

5.1. function value
followed by the value of the function in 2 to 4 alternative formats (in the same line), if some value has been assigned to the procedure identifier, or

5.2. no value assigned
otherwise.

Possible misinterpretation

5.3. The assigned function value merges with the local variables, if there is no program point in the block (see 6. Program points and 7. Heading: variables). In such a case, the value becomes erroneously the first variable of the block.

The conception of a program point will be useful further:

< program point >::= <track number> <track relative address> <stack reference> <r>

<static program point description> is a <program point> with <stack reference> equal to a BLANK.

<track number> is the relative track number of the program point, the last one being equal to 1023.

<track relative address> is the track relative address of the program point, ranging from 0 to 39.

< stack reference > indicates to a dynamical level of the program point.

< stack reference > of the program point is always equal to the stack reference of the block, where it is located.

< r > is character r, if the program point refers to the right half word instruction, otherwise < r > is a BLANK.

6. Program points (in the order of their declarations in the block):

6.1.	no type procedure	< static program point description >
6.2.	integer proc no par	-
6.3.	real proc no par	-
6.4.	Boolean proc no par	-
6.5.	integer procedure	-
6.6.	real procedure	-
6.7.	Boolean procedure	-
6.8.	switch	-
6.9.	label	-

The stack reference, constituting the complete description of the program point together with the track number and the track relative address, is not printed as it is equal to the stack reference of the block in question.

Points 6.2, 6.3, 6.4 are related to the declared functions without parameters. Points 6.5, 6.6, 6.7 are related to those with parameters. Point 6.1 is related to the declared procedure (not function) with or without parameters as well.

The notations 6.8 and 6.9 speak for themselves.

The program points are followed by simple variables and arrays in accordance with the order of their declarations in the block:

7. Heading: variables

is followed by the values of the variables represented in 2 to 4 alternative formats, applied in the following order:

⟨-d.ddddddd$_{10}$-ddd⟩	for suspected reals (not used for non-normalized values),
⟨-dddd dddd dddd⟩	for suspected integers (compulsory),
4 bytes	for suspected bit patterns (not used, if the absolute value of the variable, cosidered as an integer, is not greater than 1023),
true or false	for suspected boolean values (compulsory).

Each variable occupies one line.

Boolean variables connected with core code pieces are of the special form:

7.1. core code <first instruction addr> of <number of instructions> words with the apparent meaning.

Particularities

Working locations of the block are considered as the variables. They are the first variables of the block. The number of working locations is unknown.

Possible misinterpretations

7.2. See 5.3.

7.3. See 8.1.

8. Heading: <type> array <first element addr> <subscript ranges>
is related to an array segment.
<type> is Boolean, integer or a BLANK - for real arrays.
<first element addr> is the absolute address of the first element of the array (in the core or in the buffer). It is used for the reference to the array as an actual parameter of a procedure statement (see 9.7).
<subscript ranges> indicates to the number of values taken by each of the indexes. These numbers are separated by character × and enclosed into brackets [ ].
<type> and <subscript ranges> are not printed for subsequent arrays of the segment.

The heading mentioned above is followed by the values of the array elements or by the array picture depending on the mode of applying program ANALYSIS (see Table 1).

Possible misinterpretation

8.1. The structure will include too many subscripts, if in the block head the array declaration is followed immediatly by integer-valued variables, which are exact divisors of the number of values taken by the first subscript, the last divisor becoming the first subscript range, the previous divisors becoming the latest subscript ranges.

June, 1968
Gier Algol 4

Formats for array elements

The values of array elements are represented with the formats described in Table 1.

Each element of a Boolean array occupies a single place in the array picture.

That of an integer or real array is represented by two characters, the first being a sign (a BLANK for a non-negative value, - (minus) for negative one), the second indicating to the range of its absolute value in accordance with the subtables below.

Table 1.

type	value	picture	
Boolean	*true*: 1023 1023 1023 1023 or another 4 bytes, the first being greater than 511.	t	
	false: 0 0 0 0 or another 4 bytes, the first being less than 512.	f	
	3 elements per line	5 groups of 10 elements each per line	
		Absolute value of an element, $\|x\|$	Denotation
integer	Layout: ⟨-dddd dddd dddd⟩ is used.	$0 \leq \|x\| < {}_{10}1$	<sign>$\|x\|$
		${}_{10}1 \leq \|x\| < {}_{10}2$	<sign> a
		${}_{10}2 \leq \|x\| < {}_{10}3$	<sign> b
		${}_{10}3 \leq \|x\| < {}_{10}4$	<sign> c
		${}_{10}4 \leq \|x\| < {}_{10}5$	<sign> d
		${}_{10}5 \leq \|x\| < {}_{10}6$	<sign> e
		${}_{10}6 \leq \|x\| < {}_{10}7$	<sign> f
		${}_{10}7 \leq \|x\| < {}_{10}8$	<sign> g
		${}_{10}8 \leq \|x\| < {}_{10}9$	<sign> h
		${}_{10}9 \leq \|x\| < {}_{10}10$	<sign> i
		${}_{10}10 \leq \|x\| < {}_{10}11$	<sign> j
		${}_{10}11 \leq \|x\| \leq 549755813887$	<sign> k
	3 elements per line.	2 groups of 10 elements each per line.	

type	value	picture	
		Absolute value of an element, $\|x\|$	Denotation
real	Layout: $\{$-d.ddddddd$_{10}$-ddd$\}$ is used.		
		$_{10}{-1} \le \|x\| < 1$	<sign> a
		$_{10}{-2} \le \|x\| < {}_{10}{-1}$	<sign> b
		$_{10}{-3} \le \|x\| < {}_{10}{-2}$	<sign> c
		$_{10}{-4} \le \|x\| < {}_{10}{-3}$	<sign> d
		$_{10}{-5} \le \|x\| < {}_{10}{-4}$	<sign> e
		$_{10}{-6} \le \|x\| < {}_{10}{-5}$	<sign> f
		$_{10}{-7} \le \|x\| < {}_{10}{-6}$	<sign> g
		$_{10}{-8} \le \|x\| < {}_{10}{-7}$	<sign> h
		$_{10}{-9} \le \|x\| < {}_{10}{-8}$	<sign> i
		$_{10}{-10} \le \|x\| < {}_{10}{-9}$	<sign> j
		$_{10}{-100} \le \|x\| < {}_{10}{-10}$	<sign> k
		$7.458_{10}{-155} \le \|x\| < {}_{10}{-100}$	<sign> m
	Non-normalized values are represented by the string: $\{$<_______N_______$\}$	non-normalized	N
		0 (nil)	0
		$1 \le \|x\| < {}_{10}1$	<sign> o
		$_{10}1 \le \|x\| < {}_{10}2$	<sign> p
		$_{10}2 \le \|x\| < {}_{10}3$	<sign> q
		$_{10}3 \le \|x\| < {}_{10}4$	<sign> r
		$_{10}4 \le \|x\| < {}_{10}5$	<sign> s
		$_{10}5 \le \|x\| < {}_{10}6$	<sign> t
		$_{10}6 \le \|x\| < {}_{10}7$	<sign> u
		$_{10}7 \le \|x\| < {}_{10}8$	<sign> v
		$_{10}8 \le \|x\| < {}_{10}9$	<sign> w
		$_{10}9 \le \|x\| < {}_{10}10$	<sign> x
		$_{10}10 \le \|x\| < {}_{10}100$	<sign> y
		$_{10}100 \le \|x\| < 1.341_{10}154$	<sign> z
	3 elements per line	2 groups of 10 elements each per line.	

9. Heading: procedure call < program point >
is the description of the return point from the procedure. It is followed by the actual parameters, if there are any.

The actual parameters called by value after evaluation lose all their features. All that is known about them is their values. Thus, those that were already evaluated before the emergency are printed out in the following form:

9.1. value
followed by the value of the actual parameter in 2 to 4 alternative formats in the same line (see 7).

The actual parameters called by name and those called by value, which were not evaluated before the emergency happened, maintain their characteristics and are printed out in the following form:

9.2. Constants are represented by values in formats appropriate to their types:

9.2.1. integer constant is printed with layout {-dddd dddd dddd}

9.2.2. real constant is printed with layout {-d.ddddddd$_{10}$-ddd}

9.2.3. Boolean constant is represented by boolean value: true or false

9.2.4. string constant is represented: by:
its value - for short and
<track number> <track relative address> - for long ones,
which indicate the first string character on the drum.

Simple variables are given in the form, as follows:

9.2.5.	integer	{-dddd dddd dddd}	- for integer variables
9.2.6.	real	{-d.ddddddd$_{10}$-ddd}	- for real variables with normalized values, or
	real	N	- for those not normalized,
9.2.7.	Boolean	true	- for the true boolean variables
	Boolean	false	- for the false boolaean variables
9.2.8.	string	<track number> <track relative address>	- for string variables independant on the short or long kind.

Labels are printed in the form:

9.2.9. label <program point>
giving the complete description of the program point corresponding to the label.

Subscripted variables are represented in accordance with their types as follows:

9.2.10.	integer sub	<program point>
9.2.11.	real sub	-
9.2.12.	Boolean sub	-

which indicate to entry points for the thunks evaluating them. (The thunk is the coding implementing the evaluation of an actual parameter which is a compound expression. It is considered to be of the same level as a procedure call).

Other expressions are of the following forms:

9.2.13.	integer expr	<program point>
9.2.14.	real expr	-
9.2.15.	Boolean expr	-
9.2.16.	string expr	-
9.2.17.	design expr	-

corresponding to the thunk entries for evaluation of integer, real, boolean, string and designational expressions respectively.

An array identifier as an actual has one of the following forms:

9.2.18.	integer array	<first element address>	<number of subscripts>
9.2.19.	real array	-	-
9.2.20.	Boolean array	-	-

in accordance with the type of the actual array.

Procedure identifiers are represented by their entry point descriptions of the following forms:

9.2.21.	integer proc	<program point>
9.2.22.	real proc	-
9.2.23.	Boolean proc	-
9.2.24.	no type proc	-

A switch identifier is of the form:

9.2.25. switch <program point>

It is considered as a procedure of the fourth type - label.

References to thunks evaluating actual parameter expressions are executed:

a) Upon the very entry to the procedure body (before reservations of core code pieces in the stack and array elements in the core or buffer store) - for all parameters called by value.

b) At every point in the procedure body, where a formal called by name occures, provided that the corresponding actual is a compound expression.

Return points from the thunks are stored in the stack and printed out in the following form:

10. thunk return <program point>

indicating the point in the procedure body immediately following the call of the thunk.

Conclusion

As it was suggested by P.Naur, the stack analysis should be a set of the help 3 programs. To ease debugging the algorithm it was settled to write it in Gier Algol 4 and use the compiler in experimenting. But the ready program appeared to be effective enough to be utilized as a separate service program.

The following remarks about the program are worth mentioning:

1. The program changes nothing in the core store image (more precisely in area stack on the drum). Thus, it is quite possible to apply some other debugging programs which work with the stack after program ANALYSIS.

2. The information contained in the translated program on the drum has not been utilized at all. For example, the line numbers of program segments in the original algol text and strings might be useful. The reason for this lies in the impossibility of fetching this information in some 'natural' way without destroying the contents of the core store image.

In future compiler designs it would be desirable to maintain information about the types of variables and information connecting the stack items with identifiers in the algol text of the program.

Program ANALYSIS is not absolutely reliable, though its reliability is rather high. A perfect program for the stack analysis has to have some chain of addresses allowing identification of the stack items without any heuristic recognition, as takes place in the present program.