TECHNISCHE HOGESCHOOL TWENTE

MEMORANDUM NR. 150

A PROPOSAL FOR DEALING WITH ABNORMAL TERMINATION OF PROGRAMS.

C. BRON
M.M. FOKKINGA
A.C.M. DE HAAS

NOVEMBER 1976

Department of Applied Mathematics,
Twente University of Technology,
P.O. Box 217, Enschede, The Netherlands.

ONDERAFDELING DER TOEGEPASTE WISKUNDE

Contents

0. Abstract

In this paper a proposal is made for a mechanism to control the abnormal termination of the execution of programs or parts of programs. The notational conventions introduced are such that programs in which the possibility of abnormal termination is taken into account can be derived from the original programs without disrupting the original program's structure.
The mechanism is powerful in the sense that it allows the termination of blocks which are unknown to the blocks from within which this termination is caused. **The formal semantics closely reflect the** intuitive appeal of the construct and are similar in nature to the semantics of procedure call.

1. Definition of the concept of abnormal termination.

There is a general understanding that in program-composition the programmer's restriction to control-structures which have simple semantic properties is of beneficial value to the programmer's

product, i.e. his program.
Assuming - for the sake of the argument - the programs that we have in mind, to be semantically correct, then we still have to consider their correctness in terms of assertions about the inputs that may eventually be supplied. If some input does not fulfill the assertions we may either expect undefined behaviour of the program under execution, or preferably we would like the program to generate at least a warning about the incorrect input, after which it may terminate in - what we will call - an abnormal way.
Dijkstra terms programs which check the validity of the (input) assumptions - even if they are explicitly stated in the program-specification: robust programs, and he associates a conditional statement, composed of guarded commands with such robustness. [1]
Preferably then, our programs should be of the form:

```
determine validity of input assumptions;
if assumptions fulfilled
   then actual program else generate warning
fi
```

More often than not, it is the case that checking of the input assumption requires a sort of pre-execution of at least part of the program, which will subsequently have to be re-executed on establishment of the validity of the assumptions. Therefore it is practice to have a means of program termination that can be invoked even if execution has not reached the last statement of the program.
For the conditional statement just mentioned it is postulated that if all the guards are false, the execution of the program which contains the conditional statement is "aborted". And such an abortion may be invoked from any location in the program text.
Two typical examples of such an untimely termination could be:

1) a compiler reaching the end of the source file before it has compiled a complete program
2) a numerical program that establishes the singularity of a matrix which somehow has been derived from the input, after an attempt at inversion. Further processing is now impossible.

It is the actions that take place when in the course of processing it is found out that the input-assumptions are not fulfilled, that we will term:

"abnormal termination"

2. Partial termination

It seems unwise to consider abnormal termination of a program as a kind of magic operation that "kills" programs in execution, whilst in mid-air. After all, we may consider any user-program as a procedure which - upon having been entered into the system - is called from the operating system environment. Even if an execution is terminated abnormally, we are aware that control is returned to the calling environment, possibly leaving some indication as to the way execution was terminated. The calling environment then performs the necessary clean-up actions like the closing of files, many of which it has to do regardless of whether abnormal termination occurred or not. To put it in other words: for the calling environment it is rather irrelevant to know whether the user program has terminated normally or abnormally. This fact may, but does not need to be, part of the interface between these two environments.
Also, this view makes clear that even the "abortion" of the execution of a user-program, must be seen as a partial termination, i.e. the termination of a program-component, which - seen from the inside - appears to be "abnormal", but which - from the outside - cannot be distinguished from a "normal" termination.
We now stipulate that this view of partial termination is not only valid at the interface between an operating system and a user-program, but equally so at any interface between levels in a user program.

To be more specific: it frequently occurs that a program is designed to process a stream of data in which the outcomes of the processing of the individual data-items are rather independent of one another. (We will not attempt to define such independency in a formal way).

In processing such a stream of data we may either specify the input assertions for the stream as a whole, or for each item of the stream

individually. In the former case we expect processing to be terminated as soon as it is found out that the input assertions are not fulfilled. In the latter case an item for which the assertions are not fulfilled may be "processed symbolically". e.g. an error message is generated and if necessary the item is "skipped". Clearly in the latter case we expect abnormal termination to be restricted to the action "process one item".

3. Requirements for a descriptive mechanism.

We set out to investigate a descriptive mechanism that would fulfill the following requirements:

a) It should give the writer of a program explicit control over the concept of "abnormal termination" .

b) It should be sufficiently restrictive in form and semantics that it could not be considered as "just another control-structure".

c) It should allow for the termination of blocks from within blocks that are non-local to the block being terminated.

d) It should allow for the treatment on equal footing of "errors" which become manifest to the system-environment (e.g. overflow, index out of bounds) and "errors" which are explicitly detected in the user-written program text.

e) It should comply with the general principle of block-structured programming languages that procedures can be written, using a nomenclature which can be chosen independently from the choice in other procedures with which they may co-operate.

f) (and this was our foremost goal)
It should be possible to write programs based on the input-assertions, in which provisions for dealing with incorrect input do not influence the structure of the original program. In other words: program-specifications may be relaxed to cater for incorrect input, by adding to the program text, instead of bringing about alterations.

g) The semantics of the mechanism should be formalized and its formalization should reflect the inherent simplicity, if the mechanism is to be of practical use. (We take the simplicity of the semantics of a programming construct as a measure for the ease of its intuitive use).

h) In an implementation, the cost of catering for abnormal termination must be located in those program-components where it is potentially used, and should not range over - for instance - the cost of the procedure call/return mechanism.

i) The user of an escape mechanism should be made aware of the concept of abnormal termination on the one hand, and his obligation to specify the semantics of the block to be terminated abnormally, on the other hand.

(We must confess that the latter two requirements showed up in the course of our work, and had not been formulated beforehand .)

An evaluation of existing proposals in this field led to the conclusion that no proposal met all of the above requirements. We will not exhaustively mention all of the literature (for a fuller exposé the reader is referred to [4]), but just pick out a few examples.

- The ALGOL 60 goto-statement certainly does not meet requirement b) . Requirement c) could only be met by making affluent use of the label-parameter mechanism, and even then it would not comply with d).
- The proposed and implemented exit-constructs in BLISS [7] do not restrict themselves to "abnormal termination". It appears that untimely exit from a loop has been made a more central theme than termination of a block. Requirements c) and d) are not met.
- The mechanism which comes closest to our proposal is probably the PL/I ON-construct, although that is not restricted to the termination of blocks. One might make a restricted use of the construct, which could then, on account of the possibility of redeclaration, be compatible with e).
- The kind of control we have in mind can be made with a more general concept of labels, as in ALGOL 68, where label-variables exist. Construction of an exit-mechanism from this concept is, however, rather clumsy. And again the label-mechanism is too unrestricted. Furthermore,the label-concept may well be hostile to h) (although this point is of minor importance).
- Randell's recovery blocks [6] serve a different purpose. They provide a form of reversability to program execution such that upon malfunctioning of some kind, operations may be retried.
- Goodenough's proposals [5] come rather close to the current one, but are rather baroque, at the same time violating criterion g). Since he fails to define the scope of names that identify exceptions it is not clear whether criterion e) is fulfilled. His examples mostly

illustrate an abuse of the control-structures he proposes, where normal control would have sufficed. Furthermore, when in a procedure R an exception is raised, and R was called by Q, and Q was called by P, it is impossible to indicate the termination of P on account of the exception in R, and therefore Goodenough's proposal doesn't satisfy criterion c).

4. A proposal for an escape mechanism.

We define the environments that can be untimely terminated to be blocks (in the usual sense). This is hardly a restriction since small grains of program text can be made into blocks if necessary. The obligation to do so is a consequence of requirement i).

A block is a candidate for untimely termination if - local to that block - a so called escape procedure is declared.
An escape procedure can only be activated during the lifetime of the block in which it is declared. If an escape procedure is activated, the execution of its body replaces the execution of (the remainder of) the block in which it was declared. The execution of the body of the escape procedure takes place in the normal context of its declaration.
Activation of an escape procedure may take place from within any procedure which can be active during the lifetime of the block in which the escape procedure is declared.
The correspondence between call and the called procedure is established by matching identifiers. In order that this matching process be uniquely defined we require the declaration of an escape variable (that bears the same name as the escape procedure) in a block that surrounds both the site of activation and the site of declaration of the escape procedure.
We allow more than one escape-procedure to be bound to the same escape variable, but this binding takes place dynamically, in a nested fashion (viz. by processing of the declaration of the escape-procedure).

The escape procedure that will be activated is the one which has been most recently bound to the escape variable.
If no escape procedure is bound to an escape variable we will interpret activation of the escape variable as "program abortion".

In other words: every escape variable will be initially bound to a standard escape procedure declared outside the program. For system defined error occurrences (such as overflow, reading beyond end of file, etc.) we will assume standard escape variables with predefined names to be declared at a level surrounding the program block.
Comparing this with the way in which standard procedures are treated, we find that this proposal in no way interferes with the programmer's freedom to introduce local names in a program.

The explicit introduction of escape variables makes the proposal consistent with requirement e) of the previous chapter.
When an escape procedure is invoked, it may be supplied with parameters, which must necessarily be input parameters (value parameters). For each declared escape variable the structure of the parameter-list must be fixed. Therefore, one might consider a declaration of an escape variable together with its formal parameterlist.

There is no objection in having an escape-procedure invoking another escape procedure. The first escape procedure that terminates "normally" determines the block that is terminated abnormally.

5. Formal semantics of the escape mechanism.

In this section we will discuss in a rather informal way the formal rules which are treated extensively in a separate publication [3]. We only intend to stress the similarities and differences with normal procedures and to indicate the way in which an informal correctness proof should proceed. Hence we avoid formalization of any aspect not characteristic of the escape mechanism. Thus we assume all declared identifiers to be distinct - except for the escape procedure declarations - and we assume the procedures to be parameterless. Further, the language is supposed to be an ALGOL 60-like language, where each name should be declared before it is used. We use the conventional formula {R} stmnt {S} with the following interpretation: "if after execution of stmnt control has returned and is about to execute the textually succeeding statement, then S is true provided R is true before". Thus taking S = false implies that control does not return (either because of nontermination

or because of abortion (= control does not return anywhere in stmnt)).

Let us now consider procedure declarations. Without the escape mechanism we have the following rule:

in order to prove the behavior {R} proc p; body; restofblock {S} one should (i) prove the correctness of restofblock, i.e. {R} restofblock {S}, assuming some semantics of p, say {P} proc p {Q}, and (ii) prove the assumption made on p from the body i.e. {P} body {Q}, again assuming {P} proc p {Q} for the recursive calls inside body.

Thus formally

(i) {P} proc p {Q} $\vdash$ {R} restofblock {S}

(ii) {P} proc p {Q} $\vdash$ {P} body {Q}

{R} proc p; body; restofblock {S}

Now, when we allow escape procedures in the language, we should adapt the rule in the following way.

In addition to what has been said above, we must explicitly assume some semantics for every escape procedure in the body. These assumptions then really belong to the semantics of p.

Thus formally

(i) {P} proc p with e_i on E_i (i = 1..n) {Q} $\vdash$ {R} restofblock {S}

(ii) {P} proc p {Q}, {E_i} proc e_i {false} (i = 1..n) $\vdash$ {P} body {Q}

{R} proc p; body; restofblock {S}

where E_i should not contain any variable local to body.

But for escape procedure declarations, there is one difference:

the postcondition of the procedure's semantics must be false and the postcondition of the body must be the relaxed postcondition of the block (thus guaranteeing that the block's postcondition holds when control reaches it).

Formally

(i) {E} proc e with e_i on E_i (i = 1..n) {false} $\vdash$ {R} restofblock {S_1}

(ii) {E} proc e {false}, {E_i} proc e_i {false} (i = 1..n) $\vdash$ {E} body {S_2}

{R} escape procedure e; body; restofblock {S_1 or S_2}

where E_i should not contain any variable local to body.

Note that the semantics for escape procedures are just an instance of a normal procedure's semantics.

Now, we consider procedure calls.

Without the escape mechanism in the language, we have the following rule.

upon precondition P the call certainly yields Q as postcondition - provided {P} proc p {Q} holds - and moreover, if I doesn't contain any variable used by p, then I clearly remains invariant and finally, if h is not used by p we may assert that h is just *some* hypothetical constant (occurring in P, Q and I in order to relate the initial and final values of the variables used by p).

Formally

$\{P\}$ proc p $\{Q\}$ / $\{\exists h.\ P \wedge I\}$ call p $\{\exists h.\ Q \wedge I\}$

where h nor I do contain any variable used by p.

(For example, suppose $\{x = x_o \wedge y = y_o\}$ proc p $\{y = x_o \wedge x = y_o\}$ and x_o, y_o are not used by p, then by the last rule

$\{\exists x_o, y_o.\ x = x_o \wedge y = y_o \wedge x_o = 17 \wedge y_o \leq 3\}$ call p

$\{\exists x_o, y_o.\ y = x_o \wedge x = y_o \wedge x_o = 17 \wedge y_o \leq 3\}$

so by the rule of consequence $\{x = 17 \wedge y \leq 3\}$ call p $\{y = 17 \wedge x \leq 3\}$.)

But when the semantics for p contains some assumptions about escape procedures, then in addition:

these assumptions should be verified from the currently visible procedures (as introduced by the textually nearest declarations) and of course, what is invariant over p may be added to the preconditions of the escape procedures calls.

Formally

$$\frac{\{P\}\ \text{proc p with } e_i \text{ on } E_i\ (i = 1 .. n)\ \{Q\} \qquad \{E_i \wedge I\}\ \text{proc } e_i\ \{\text{false}\}}{\{\exists h.\ P \wedge I\}\ \text{call p}\ \{\exists h.\ Q \wedge I\}}$$

where I nor h do contain any variable used by p.

Note, p itself may be an escape procedure (and then Q = false). In order to prove the n premises above the line from the currently available semantics for the e_i we have the following rule:

The (stronger) precondition we want to have for e_i should imply the (weaker) one we already have (from the nearest declaration), and further, the assumptions on their escape procedures should be verified in the same way. (Note that they may be recursively dependent).

Thus

$$\frac{\{E_i\}\ \underline{\text{proc}}\ e_i\ \underline{\text{with}}\ e_{ij}\ \underline{\text{on}}\ E_{ij}\ (j = 1..n_i)\ \{\text{false}\}\ (i = 1..n) \qquad E'_i \vdash E_i\ (i = 1..n)\ ,\ E_i \vdash I_i\ (i = 1..n) \qquad \{E'_i\}\ \underline{\text{proc}}\ e_i\ \{\text{false}\}\ (i = 1..n) \vdash \{I_i \wedge E_{ij}\}\ \underline{\text{proc}}\ e_{ij}\{\text{false}\}(j=1..n_i, i=1..n)}{\{E'_i\}\ \underline{\text{proc}}\ e_i\ \{\text{false}\}\ (i = 1..n)}$$

where I_i does not contain any variable used by e_i.

In practice we do not expect escape procedures to be recursively dependent.

Finally, we consider escape variable declarations. For simplicity we choose the following way

new e : escape; restofblock

should be replaced by

new e ; escape procedure e ; abort ; restofblock.

Then, new e has no semantic effect and further, abort is meant to be an escape procedure call after which control does not return anywhere in the user program. Thus it is axiomatized by {true} abort {false}.

6. Implementational Aspects

With regard to the implementation of escape variables and procedures we note that the proposed mechanism has a great deal of similarity to mechanisms for the handling of formal procedure parameters.
Here, the escape variable plays the role of the formal procedure parameter, and the declaration of an escape procedure plays the role of binding the actual procedure parameter to the formal one.
Unlike the procedure parameter mechanism, however, to each escape variable a stack of escape procedures may be bound of which the top element is the currently valid binding.

From the above we derive that an implementation of escape-procedures and variables has to cater for the following:

- Escape variables will have to be initialized (upon entry into the block in which they are declared) in such a way that the causing of an escape will result in program abortion if no escape procedure has been bound to the escape variable.
- For each declaration of an escape procedure a data-segment has to be created, containing:
  a) the environment and address of the escape procedure itself.

b) the return information of the block in which the escape procedure is declared, since this will also function as the escape procedure's return information.

c) the data for maintenance of the stack of bindings to the corresponding escape variable.

- Upon normal exit from a block in which escape procedures have been declared the bindings with regard to the corresponding escape variables will have to be restored to their prior value
- causing of an escape proceeds in a manner entirely analogous to the calling of a formal procedure: the escape variable is bound to the escape procedure whose data-segment contains the environment and address information
- upon exit from an escape procedure, several block-instances may have to be terminated simultaneously. Re-establishing the return-context implies that also the bindings of escape procedures to escape-variables will have to be revised. (There is at least one escape-variable for which this is the case). A convenient way to implement this obligation is to have all escape procedure data segments chained in reverse order of their creation. If the segments have been allocated stackwise in a linear address space it is easy to break down that chain up to the return-level.
  As a matter of fact, the same mechanism can be employed for exit from a block in which escape procedures have been declared. The fact that here only one block is terminated is not of importance.

From the above considerations it will be clear that the runtime cost incurred by the proposed escape mechanism is entirely located in:

- entry to a block in which an escape variable is declared (In practice this will be the sort of block with a long lifetime)
- entry to, and exit from a block in which escape procedures have been declared.
- the actual occurrence of an escape situation.

We therefore conclude that requirement h) of chapter 3 is indeed met by the current proposal.

7. Examples of the use of escape procedures

Rather than to give a number of programmed examples of the use of escape procedures we will discuss a few real-life situations in which the proposed construct could play a vital role. Realistic examples in the form of programs tend to become too large or too cluttered with details as to be of illustrative value.
Two examples stem from concrete programming projects, viz. an interactive LISP interpreter and an ALGOL compiler.

In the LISP interpreter we wanted to deal in a different manner with two kinds of errors.
If during processing of a LISP-form a syntactic (or semantic) error was detected, processing of the current form had to be abandoned, the remainder of the form on the input or in an input buffer had to be skipped, an error-message had to be generated and the interpreter should return to a state in which it is ready to accept a new form. (Note that the result of previous forms in which functions were defined had to be preserved).
Therefore, local to the procedure READANDINTERPRETLISPFORM, an escape procedure SYNTAXERROR must be declared which can be activated from within any of the syntactic routines that are local in READANDINTERPRET-LISPFORM.

If (even after garbage collection) list memory is exhausted, the LISP processor may (after suitable clean-up actions) return to a state with an empty list memory in which it is - once again - ready to accept input. Therefore, local to the procedure ACCEPTLISPFORMS an escape procedure MEMORYFULL must be declared. Upon termination of ACCEPTLISPFORMS it may be decided to reinitialize the LISP-interpreter with an empty list-memory.

Note that in both cases the form of control required could have been obtained by means of a goto statement. This is not the case in the following example:
In the THE-ALGOL Compiler [2] a number of tables have been declared to store identifiers, block structure data, nesting of if-then-else constructs and others. These tables were predeclared of a size suitable to cater for the compilation of even large-sized programs,

but they might give rise to table overflow in certain pathological cases. Rather than inserting size-checks on all operations that expanded the contents of the tables, we resorted to the index-checking mechanism which was a non-optional software feature (it might equally well have been a hardware check like in the Burroughs B6700). To the user it would be unpleasant to be confronted with the dynamic error message "index out of range" if - to all his knowledge - his program wasn't even in execution, but in the compilation phase. At the time, the problem was overcome in a very ad-hoc way.

The possibility of linking an escape procedure to the standard escape variable: "INDEXERROR" would have solved the problem in a natural way. Furthermore, by declaring one or more instances of the escape procedure on a sufficiently local level, the table responsible for the overflow could have been pinpointed, after which a call on a more outward declared escape procedure could still have caused termination of the compilation. Analogously, the very same error caused in the indexing of tables that were not supposed to give rise to overflow could have been detected and reported as a "compiler error", by binding another escape procedure to INDEXERROR at a suitable level.

Finally, and this remark seems almost superfluous, the proposed escape mechanism exactly reflects what we have in mind when we say that the execution of a program is aborted: i.e. the program block returns control to the calling environment after having performed the necessary actions. The standard escape procedure ABORT may not only generate a message, but it is able to provide a stack dump of any form of sophistication, since its body is executed before any environment is terminated abnormally.

The last example below gives the text of a (more or less complete program) in which standard escapes (= "system errors) are exploited in a variety of ways. The example contains a proof in the formalism of [3]. Those not familiar with the formalism may erase everything within and including the curly brackets { and }, and will still be able to appreciate the remaining text.

We use the following notation:

The symbol "letrec" opens the scope of a set of declarations which ranges over the declarations proper and the subsequent statement sequence enclosed between brackets "in" and "ni" .

In this example we assume that the environment contains:

```
    OF : esc
    {x = x_o} recip ({with esc OF on maxreal < 1/abs (x_o)}
                    var res val x {hyp x_o}) {res = 1/x_o}
```

The program then may read as follows

```
(* here is some innerblock of the program : *)
letrec var sing : esc;
       proc {M = M_o} invert ({with esc sing on M_o is singular}
            var I val M {hyp M_o}) {I = M_o^-1} =
            letrec esc proc OF = (* just escape by calling *) sing
                in "standard way of computing the inverse
                    of the initial M using the procedure recip
                    for reciprocal"
                ni (* end of procedure invert *)
    in (* stmnt part of the innerblock *)
       (* we want to print the inverses of a series of matrices to be read in *)
       for i := 1 to n
       do letrec esc proc {MX is singular} sing {false} =
                  print ("matrix no", i, "is singular")
              in read (MX);
                 {∃M_o. MX = M_o ∧ MX = M_o}
                 invert ( {the current environment, including MX = M_o
                           implies {M_o is singular} sing {false} qed}
                          var Inv val MX)
                 {∃M_o. Inv = M_o^-1 ∧ MX = M_o};
                 {Inv = MX^-1}
                 print (Inv)
              ni {i-th inverse printed or singularity message printed}
       od {the first n matrices have been processed}
       .............
       (* here follows a computation which in no reasonable way can be
          continued upon singularity *)
          ... invert ((* program abortion if M is singular *) I, M) ...

       .............
       (* here follows a computation in which maxreal is interpreted as
        infinity *)
       read (x);
       letrec esc proc OF = (invx := maxreal) in recip (invx,x) ni; ...
    ni (* end of the innerblock *)
```

With regard to the above program the following remarks are in place:

1) within the body of invert the occurrence of overflow is interpreted as singularity of the matrix to be inverted.
2) In the statement executed under control of the for-clause there is a sensible reaction to the occurrence of a singular matrix (detected in the course of invert). If, however, overflow would be caused during the call of read or print, this is an unforeseen event, and abortion will result.
3) In the (textually) last call of invert there is no escape action defined with the occurrence of singularity.
4) In the next to last line of the example, a local decision is made to replace the result of 1/x by maxreal if the division would cause overflow.

8. Conclusion

We have suggested a language-mechanism for dealing with abnormal situations during program execution. In chapter 3 we listed requirements that - in our opinion - should be met by an acceptable mechanism. These requirements were set out beforehand as a goal for our investigation. Of the many proposals on this matter which have been published in the literature we found none which satisfied all criteria.

The current mechanism, however, does. The formal semantic rules reflect the intuitive notion that the concept of escape procedures is hardly more complex than the concept of procedures as such.

One of the most attractive features of the proposal is the fact that programs written on rather strong input assumptions may be adapted to a relaxation of these assumptions by the addition of escape procedures, which in no way influence the textual structure of the original program.

9. References

[1] Dijkstra, E.W., Guarded Commands, Nondeterminary and Formal Derivation of programs.
Comm. A.C.M. 18 (1975) 453-458.

[2] Dijkstra, E.W., The structure of the THE-multiprogramming System.
Comm. A.C.M. 11 (1968), 341-345.

[3] Fokkinga, M.M., Axiomatization of Declarations: Variables, Procedures, Escape-clauses.
Twente University of Technology, to be published.

[4] Haas, A.C.M. de, Escape Clauses in Programming Languages,
M.Sc. Thesis, Twente University of Technology,
Dept. of El Engineering (1976).

[5] Goodenough, J.B., Exception Handling: Issues and a Proposed Notation.
Comm. A.C.M. 18 (1975) 683-696.

[6] Randell, B., System Structure for Software Fault Tolerance,
Int. Conference on Reliable Software (Los Angeles 1975).

[7] Wulf, W.A., Russell, D.B., Habermann, A.N. BLISS: A Language for Systems Programming.
Comm. A.C.M. 14 (1971) 788-790.