Formal Descriptions of Programming Concepts, E.J. Neuhold (ed.)
North-Holland Publishing Company, (1978)

Axiomatization of Declarations and the Formal Treatment of an Escape Construct

Maarten Fokkinga

Tech. Univ. Twente, Enschede, The Netherlands

ABSTRACT

We give Hoare-like axiomatizations for variables, procedures and a new escape construct. The proof rules treat scope rules properly; simplifications are shown for a restricted use of global variables and for a disallowance of innerblocks. It is shown that the axiomatization of the escape construct has influenced its design.

1.INTRODUCTION

In this section we give three criteria which should be fulfilled by any axiomatization and we extend the usual formulae expressing partial correctness by an environment component.

Some criteria

We feel that (some) existing axiomatic definitions are not fully satisfactory. This holds in particular with respect to the following criteria for the proofrules of the axiomatic system.
c1. The rules formalize the concept of "scope".
c2. The rules reflect and more importantly direct a (the) natural way of reasoning about the correctness of the program constructs.
c3. The rules do not necessitate any alteration of the program text and their applications can be written unambiguously within the program text.

Definitions of procedures by temporarily valid rules, viz relative to the scope of the procedure declaration as in Hoare & Wirth (1973), run the risk not to be a well understood formal system because temporal validity of proof rules is not well known in mathematical logic. The rules for procedures presented in Donahue (1976) are inconsistent because they do not treat the scope rules properly. Rules which model procedure call by body replacement conflict with c2 because procedures should be considered as abstractions rather than as abbreviations of a program text and consequently should be inspected just once. Rules which treat scope rules by renaming in the program text, Hoare (1971), Hoare & Wirth (1973), conflict with c3 (and c2). Donahue (1976) seems to be the first one explicitely requiring the first half of c3.

The environment component

The usual formalism of formulae $\{P\}S\{Q\}$ is not rich enough to give a satisfactory treatment of scope rules due to its inability to express the environment of a statement S and more importantly, to update the environment. Therefore the formulae of our axiomatic system have the format "$Env\vdash\{P\}S\{Q\}$", to be read as "in the environment Env the behaviour $\{P\}S\{Q\}$ holds". Env is a set of formulae representing semantic properties of names which are statically true in S; these properties may be described by the declarations in whose scope/range S is contained. The symbol $\vdash$

does not denote the deducibility relation but is merely used as a separation mark as in Gentzen-like calculi of sequents;the symbol $\vdash_H$ is used for deduction in the Hoare-like system. The type of formulae which may be taken into Env will be described when needed; some typical examples however are "var V: int", indicating that v is a variable of type int, and

$$\underline{\text{proc}}\{x = x_0 \wedge y = y_0\}\text{exch}(\underline{\text{var}}\ x,y{:}\text{int}\{\underline{\text{hyp}}\ x_0,y_0{:}\text{int}\})\{x = y_0 \wedge y = x_0\}$$

indicating that exch is a procedure which exchanges the value of its two parameters. In order to describe the updatings of Env the following notation is used: + and - for set theoretic addition and deletion,[xxx] for all formulae in Env in which the main identifier is tagged with xxx (xxx is var, proc, esc, ...), [xxx id] for all formulae of Env in which the main identifier is id and is tagged with xxx, and [id] for all formulae of Env in which id has a free occurrence. The notion of free occurrence is as usual; in particular in a formula expressing some semantics of a procedure, the formal parameter names, included those within the curly brackets {and}, bind the corresponding names in the entire formula. Bound names may always be renamed without affecting the meaning of the formulae. Finally, [id ← id'] denotes the usual substitution operator which substitutes id' for each free occurrence of id - taking care of bound names -; simultaneous substitution is denoted by [id1,id2 ← id1',id2'].

The environment component clearly corresponds to the environment argument in denotational semantics, Tennent (1976), and to synthesized attributes in attribute semantics, Knuth (1968). The keywords var, proc, ... correspond to injection as well as projection and inspection in a disjoint sum!
The concept of inironment has also been formalized in Clarke (1976) and Apt & de Bakker (1977). Both however treat the semantics of a procedure as a piece of program text, thus conflicting c2 and c3.

2. AXIOMATIZATION OF VARIABLES AND PROCEDURES

In this section we distinguish between various syntax properties. These strongly influence the formal rules for variables and procedures.

2.1. Unrestricted globality

The purpose of any declaration is to introduce a new entity, say with name n, and to state properties of it. So the environment must be updated with the static properties of n, nsem say, and potentially existing objects with the same name must be considered distinct.

... possibly another premise justifying nsem ...

$$D_1\quad \frac{\text{Env}[n\leftarrow n'] + \text{nsem} \vdash \{R[n\leftarrow n']\}\text{stmnt}\{S[n\leftarrow n']\}}{\text{Env} \vdash \{R\}\underline{\text{new}}\ \text{decl-of-n}\ \underline{\text{in}}\ \text{stmnt}\ \underline{\text{ni}}\{S\}}$$

where n' does not occur free in Env and n (and consequently R and S).

Some of the static properties of n, nsem, might be clear from the text (the type and the fact that it is a var or proc), others might require an additional proof (the net effect of a proc) and give rise to the other premise.

Variable declaration.
Thus the rule for variable declaration reads

$$V_1\quad \frac{\text{Env}[v\leftarrow v'] + \underline{\text{var}}\ v{:}\text{tv} \vdash \{R[v\leftarrow v']\}\text{stmnt}\{S[v\leftarrow v']\}}{\text{Env} \vdash \{R\}\underline{\text{new}}\ \underline{\text{var}}\ v{:}\text{tv}\ \underline{\text{in}}\ \text{stmnt}\ \underline{\text{ni}}\{S\}}$$

where v' not free in Env, v.

This should be compared with the usual axiomatizations as in Hoare & Wirth (1973):

$$V_2\quad \frac{\underline{\text{var}}\ v'{:}\text{tv} \vdash_H \{R\}\text{stmnt}[v\leftarrow v']\{S\}}{\{R\}\underline{\text{new}}\ \underline{\text{var}}\ v{:}\text{tv}\ \underline{\text{in}}\ \text{stmnt}\ \underline{\text{ni}}\{S\}}$$

where v' not free in R, S and, except for v≡v',
not free in stmnt either.

According to the latter rule the innerblock program writer is forced to choose new names in the course of the correctness proof whereas - hopefully - he had chosen the best names you can think of in the original program text. In the former rule however a new (read meaningless) name has to be chosen for an entity which is in the inner block much more meaningless than the variable v. Moreover, according to the latter rule the correctness proof cannot easily be written within the original program text, whereas according to the former rule we can place assertions in the program text in a natural way such that they unambiguously determine the relevant data for the formal rule:

$\ldots\{R\}\ \underline{\mathrm{new}}\ \underline{\mathrm{var}}\ v\{\leftarrow v'\}{:}tv\ \underline{\mathrm{in}}\ \{R[v\leftarrow v']\}\ \mathrm{stmnt}\ \{S[v\leftarrow v']\}\ \underline{\mathrm{ni}}\ \{S\}\ldots$

with a suitable convention of how surrounding declarations determine the environment.

Initializing variable declaration

Because the declaration has "side effects" the initial state for stmnt differs from {R[v←v']} and we also need a second premise:

$$V_3\quad \frac{\begin{array}{l}\mathrm{Env} \vdash R \supset R'[v,v'\leftarrow \mathrm{exp},v]\\ \mathrm{Env}[v\leftarrow v'] + \underline{\mathrm{var}}\ v{:}tv \vdash \{R'\}\ \mathrm{stmnt}\ \{S[v\leftarrow v']\}\end{array}}{\mathrm{Env} \vdash \{R\}\ \underline{\mathrm{new}}\ \underline{\mathrm{var}}\ v{:}tv{:=}\mathrm{exp}\ \underline{\mathrm{in}}\ \mathrm{stmnt}\ \underline{\mathrm{ni}}\ \{S\}}$$

where v' not free in Env, v.*)

Note that it is very clearly stated that the declaration is nonrecursive: exp is interpreted in Env in stead of Env [v←v'].

List of (Initializing variable) declarations

A number of objects may be declared at a time in two ways: either simultaneously, separated by commas, or sequentially, separated by semicolons. It is very important that the defined names are distinct and so the construct is not merely a nest of blocks each one having a single declaration. Indeed, in the following rules the substitutions are ill defined if v, v' do not constitute a list of distinct names and moreover the restoration of the original values of v (performed by the transition from S[v←v'] into S) is done at once for all members of v together. In the simultaneous case we have

$$V_4\quad \frac{\begin{array}{l}\mathrm{Env} \vdash R_0 \supset R_1[v,v'\leftarrow \mathrm{exp},v]\\ \mathrm{Env}[v\leftarrow v'] + \underline{\mathrm{var}}\ v_i{:}tv_i\ \ (i = 1 \ldots n) \vdash \{R_1\}\ \mathrm{stmnt}\ \{S[v\leftarrow v']\}\end{array}}{\mathrm{Env} \vdash \{R_0\}\ \underline{\mathrm{new}}\ \underline{\mathrm{var}}\ \ldots,\ v_i{:}tv_i{:=}\mathrm{exp}_i\ \ldots\ \underline{\mathrm{in}}\ \mathrm{stmnt}\ \underline{\mathrm{ni}}\ \{S\}}$$

where v' is a list of distinct names not occurring free in Env, v

and the sequential case (denoting $v_1,\ldots,v_i$ by $v_{1..i}$):

$$V_5\quad \frac{\begin{array}{l}\mathrm{Env}[v_{1..i-1}\leftarrow v'_{1..i-1}]+\underline{\mathrm{var}}\ v_j{:}tv_j(j{=}1\ldots i{-}1)\vdash R_{i-1}\supset R_i[v_i,v'_i\leftarrow \mathrm{exp}_i,v_i],\ (i{=}1\ldots n)\\ \mathrm{Env}[v\leftarrow v'] + \underline{\mathrm{var}}\ v_j{:}tv_j\ \ (j = 1\ldots i)\vdash\{R_n\}\ \mathrm{stmnt}\ \{S[v\leftarrow v']\}\end{array}}{\mathrm{Env} \vdash \{R_0\}\ \underline{\mathrm{new}}\ \underline{\mathrm{var}}\ \ldots;\ v_i{:}tv_i{:=}\mathrm{exp}_i\ldots\underline{\mathrm{in}}\ \mathrm{stmnt}\ \underline{\mathrm{in}}\ \{S\}}$$

where v' distinct and not free in Env, v.

Written within the program text the latter looks like

$\ldots\ \{R_0\}\ \underline{\mathrm{new}}\ \underline{\mathrm{var}}\ \ldots;\ \{R_{i-1}\}v_i{:}tv_i{:=}\mathrm{exp}_i\ldots\ \underline{\mathrm{in}}\ \{R_n\}\ \mathrm{stmnt}\ \{S[v\leftarrow v']\}\ \underline{\mathrm{ni}}'\{S\}.$

2.2. Restricted globality

We restrict globality in the following way. First define (recursively): a name

*) A deduction of the first premise requires many more rules which will not be given in this paper. The intuitive interpretation however is that R ⊃ R'[v,v'←exp,v] should be proved using facts from Env.

n is used globally in a text T if either n occurs free in T or n is used globally in the declaration of a procedure occurring free in T. Recall further that the range of a declaration is, informally, the text for which the declaration is valid and that the scope of a declaration is the range with exclusion of all those subranges which belong to a declaration of an object with the same name. Then define: globality is restricted if no name is used outside its scope. The restriction disallows the following typical example:

new var v; proc p = (v:=0) in ... new var v in p ni ... ni.

In general, all those programs are disallowed whose semantics will be affected by extending the formal and actual parameter list by the global variable names, or equivalently, by adding the <nomenclature> of Dijkstra (1976) to each block. With restricted globality new objects really hide existing objects with the same name and so the updating of Env should have the form Env-[n] in stead of $Env[n \leftarrow n']$. Although an outer var v is "nonexistent" in an innerblock new decl-of-v in ... ni its relation to other variables may be changed in the innerblock as shown by $\{v<w\}$ new var v in w:=w-1 ni $\{v \le w\}$. This necessitates the hypothetical constants in the scheme for declarations with restricted globality:

--- possibly another justifying nsem ---

$$D_2\quad \frac{\text{Env-}[n,n'] + \text{nsem} + n':tn' \vdash \{R\}\ \text{stmnt}\ \{S\}}{\text{Env} \vdash \{\exists n':tn'.\ R\wedge I\}\ \underline{\text{new}}\ \text{decl-of-n}\ \underline{\text{in}}\ \text{stmnt}\ \underline{\text{ni}}\ \{\exists n':tn'.\ S\wedge I\}}$$

where[*)]

a. n' is a possibly empty list of distinct names,
b. no variable used globaly in stmnt is free in I,
c. n is not free in R, S and n'.
(By conversion the range of ∃ and later on ∀ extends as far right as possible).

Thus I comprises all relations of interest whose variables can not be subject to change in stmnt (and in particular fixes the relations of n' to these variables). In other words: the transition from S via ni to $\exists n':tn'.S\wedge I$ restores the original values (better: relations) of the variables which have been made invisible within stmnt.

Although the rule might look more complicated, it actually precribes a simpler way of reasoning, because it formalizes both the invisibility of an outer n from within the block and the invariance of some relations I. Thus criterion c2 has been satisfied. When the list n' is empty, $\exists n':tn'.A$ is understood to be A, for any assertion A.

(Remark. The following adaption of scheme D1,

--- possibly another premise justifying nsem ---

$$\frac{\text{Env-}[n] + \text{nsem} \vdash \{R[n \leftarrow n']\}\ \text{stmnt}\ \{S[n \leftarrow n']\}}{\text{Env} \vdash \{R\}\underline{\text{new}}\ \text{decl-of-n}\ \underline{\text{in}}\ \text{stmnt}\ \underline{\text{ni}}\ \{S\}}$$

where n' not free in Env,n

fails, because nothing is known about n' from within stmnt, whereas its type is needed in proving an implication involving n').

Here is an example of the use of (D2) for variables:

let the current environment contain var v,w:int,
$\{0<v<w\}\{\exists v':nat,(v'<w)\wedge(0<v' \vee w'=v)\}$
new $\{v':nat,\}$ var v:int in $\{v'<w\}$ v:=w;v:=v-1; w:=v $\{v'\le w\}$ ni
$\{\exists v':nat.\ (v'\le w)\wedge(0<v'\wedge v'=v)\}\{0<v\le w\}$.

[*] The rule requires the pre- and postconditions to be of a particular format and even prescribes the choice for the bound names n'. This strange feature can be eliminated by replacing the pre- and postconditions by Ro and So and adding the premises $Env \vdash Ro \supset (\exists n':tn'.R\wedge I)$ and $Env \vdash (\exists n':tn'.S\wedge I) \supset So$.

Procedures with restricted globality

Consider the following syntax for recursive procedure declarations: new proc p (spec f:tf)=body in stmnt ni, where spec f:ft stand for a list of specifications, viz. var x:tx and val y:ty, indicating the Pascal variable and value parameter mechanism. Assume further that globality is restricted. A natural way of specifying some semantics of a procedure p, psem say, is the format proc {P} p(spec f:ft{hyp h:th}){Q} where P and Q describe a net effect of p in terms of the formal parameters f, the hypothetical constants h and the global variables of p, i.e. the variables globally used in the declaration of p. Note that the global variables are precisely the free variable names in psem. The formal val parameters are considered as *local variables* and therefore may not occur free in Q (restr.1). The global variables are considered as *implicit* var-parameters and must have a free occurrence in the entire formula in order to have a consistent and concise interface between the rules for declaration and call (restr.2). The hypothetical constants may be used to relate the initial and final values of the (implicit and explicit) var-parameters, e.g. proc $\{x=x_0-1\}$ incrx2({hyp x_0:int})$\{x=x_0+1\}$ and must of course be distinct from f (restr. 3). In the sequel, these three restrictions are tacitly assumed. We assume in addition the following syntax property: in each call *all var-parameters* (i.e. the explicit actual var parameters and the variable names free in psem) are distinct.

So let psem abbreviate proc {P}p(spec f:tf {hyp h:th}){Q}.
The rule for procedure declaration then reads

$$P_1\quad \frac{\begin{array}{l}\text{Env-[p] + psem - [f,h] + var f:tf + h:th} \vdash \{P\}\ \text{body}\ \{Q\}\\ \text{Env-[p,p'] + psem + p':tp'} \vdash \{R\}\ \text{stmnt}\ \{S\}\end{array}}{\text{Env} \vdash \{\exists p'\text{:tp'}.R\wedge I\}\ \text{new proc p (spec f:tf) = body in stmnt ni}\ \{\exists p'\text{:tp'}.S\wedge I\}}$$

where the restrictions on p', R, S, I are obvious from D2.

And the rule for procedure call is an axiom rather than a rule; it closely resembles the rule for blocks:

PC1: Env + psem $\vdash\{\exists h\text{:th}.P[f\leftarrow a]\wedge I\}$ p(a)$\{\exists h\text{:th}.Q[f\leftarrow a]\wedge I\}$
where *all var-parameters* not free in I.

Again, I comprises all relations of interest whose variables are not subject to change within procedure p. The way one actually reasons (should reason?) about procedures might be slightly better reflected in these rules than e.g. those in Hoare (1971) and Hoare & Wirth (1973). The rules have moreover the advantage that their application can unambiguously be written in the program text, see appendix A.

The following examples show that it is really necessary to have a means of deleting information from the environment - if no alteration of the program text is allowed -:

new proc zero = (y:=0) in new proc zero=(y:=1) in zero ni ni,
new proc zero = (new proc zero = (y:=1) in zero ni) in zero ni,
new proc zero = (y:=0); proc p = (new proc zero=(y:=1) in zero ni) in p ni.

For each of these blocks the behaviour {true}--{y=0} might wrongly be provable, whereas {true}--{y=1} is right. Thus the rules presented in Donahue are inconsistent, the failure in the soundness proof being a wrong translation from the axiomatic formula into the denotational one.

Remark
One might feel it more natural to use the formal val-parameters in the post-condition of psem while expressing their initial values. Consider for example (a) proc $\{y=y_0\}$ assign (var x val y {hyp y_0})$\{x=y_0\}$ with (b) proc {true} assign (var x val y) {x=y}. But because we assume the PASCAL val-parameter mechanism, a call like assign(v, v+1) is allowed. Therefore we feel (b) slightly misleading. It is however quite well possible to allow (b), but then the rules have to force the introduction of additional hypothetical constants, y_0 say, in the justification

of not only psem but also the correctness of a call of p. It will appear that the correctness proofs as directed by these rules will be quite the same.

Disallowing innerblocks

The language PASCAL disallows innerblocks. All local entities must be declared at the entry of a procedure body (hence the keyword new is superfluous). This may look rather inconvenient and one may wonder whether any *semantic* simplification has been gained. It turns out that the rule for declarations may be simplified if in addition the following, quite reasonable, syntax property is assumed for each procedure *the {list of distinct} locally declared names are distinct from the formal parameter names* and also *distinct from the global variable names* {which due to the restriction on globally is certainly satisfied when there are no initializing variable declarations}. They need not necessarily be distinct from the formal parameter names and local names of the local procedures. Here is the scheme for the justification of a procedures semantics:

$$\text{D3}\quad \frac{\begin{array}{l}\text{--- possibly another premise justifying } n_i\text{sem ---}\ \ ,\ (i=1...m)\\ \text{Env-}[n] + n_i\text{sem } (i=1..m) \vdash \{R'\}\ \text{stmnt}\ \{S\}\end{array}}{\text{Env} \vdash \{R\}\text{decl-list-of-n } \underline{\text{in}} \text{ stmnt } \underline{\text{ni}}\ \{S\}}$$

(where n not free in R,S).

(R' equals R except when there are initializing variable declarations.) Thanks to the restriction "n not free in R and S" the rule is also sound when innerblocks are allowed or the distinctness property does not hold; but then the rule is no longer complete, as shown by the impossibility to prove by (D3)

{v=0} new decl-of-v in --- ni {v=0} and
proc {x=0}p(var x:int){x=0} = (var x:int in skip ni).

Indeed, actually each name in the precondition R of the *procedure body* is necessarily a formal or global parameter, hence distinct from n, and therefore cannot be made invisible and consequently needs not to be "restored" upon passing the closing ni.
Here we specify (D3) for the case that the list of declarations is a *sequential* one of initializing variable declarations followed by procedure declarations

$$\text{P}_2\quad \frac{\begin{array}{l}\text{Env-}[v,p] + \underline{\text{var}}\ v_j{:}tv_j (j=1..i-1) \vdash R_{i-1} \supset R_i[v_i \leftarrow exp_i] \qquad ,(i=1..m)\\ \text{Env-}[v,p] + \underline{\text{var}}\ v_j{:}tv_j (j=1..m) + p_j\text{sem}(j=1..i)\\ \qquad -[f_i,h_i] + \underline{\text{var}}\ f_i{:}tf_i + h_i{:}th_i \vdash \{P_i\}\text{body}_i\{Q_i\}, \quad (i=1..n)\\ \text{Env-}[v,p] + \underline{\text{var}}\ v_j{:}tv_j (j=1..m) + p_j\text{sem}(j=1..n) \vdash \{R_n\}\text{stmnt}\{S\}\end{array}}{\text{Env}\vdash\{R_0\}\underline{\text{var}}...;v_i{:}tv_i{:=}exp_i...;\underline{\text{proc}}...;p_i(\underline{\text{spec}}\ f_i{:}tf_i)=\text{body}_i...\underline{\text{in}}\ \text{stmnt}\ \underline{\text{ni}}\{S\}}$$

(where x,p not free in R_0,S).

Thus the proof of the procedure body breaks down to (m+)n+1 independent subproofs.

2.3. Procedures with unrestricted use of global variables

The main problems caused by the use of global parameters - in their full glory - is that distinct objects may be known under the same name: how then to describe the semantics of a procedure? Consider for instance the declaration
proc p (var x:int)=zero in an environment which contains var x:int and
proc {true} zero {x=0}.
It seems that a change in the syntax of the language is necessitated in order to obtain a definition satisfying the criteria. Thus assume that a declaration of procedures looks like proc p(spec:tf) = spec f:tf in body corp (cfr Algol 68) where only the type of the procedure is specified in its heading and not the formal parameter names. Then we are free to choose new names, f' say, as fancy formal parameters in the semantics psem so that the globals, f, can have their own name. It is only in the range of spec f:tf that the globals f should be

renamed, into f" say, and the fancy formal parameter names f' should be renamed into f:
Let psem abbreviate $\underline{\mathrm{proc}}\{P\}p(\underline{\mathrm{spec}}\{f'\}{:}tf\{\underline{\mathrm{hyp}}\ h{:}th\})\{Q\}$

$$\mathrm{P3}\quad \frac{\begin{array}{l} (Env[p\leftarrow p'] + psem)[f\leftarrow f''] + \underline{\mathrm{var}}\ f{:}tf + h{:}th \vdash \{P[f,f'\leftarrow f'',f]\}\ \mathrm{body}\ \{Q[f,f'\leftarrow f'',f]\} \\ Env[p\leftarrow p'] + psem \vdash \{R[p\leftarrow p']\}\ \ \mathrm{stmnt}\ \{S[p\leftarrow p']\} \end{array}}{Env \vdash \{R\}\ \underline{\mathrm{new}}\ \underline{\mathrm{proc}}\ p\ (\underline{\mathrm{spec}}{:}tf)=\underline{\mathrm{spec}}\ f{:}tf\ \underline{\mathrm{in}}\ \mathrm{body}\ \underline{\mathrm{corp}}\ \underline{\mathrm{in}}\ \mathrm{stmnt}\ \underline{\mathrm{ni}}\ \{S\}}$$

where p' not free in Env,p
and f", h distinct and not free in Env,p,p',f.

Again, it is easy to write it unambiguously within the program text:

$$\begin{array}{l} \ldots\ \{R\}\ \underline{\mathrm{new}}\ \underline{\mathrm{proc}}\ \{P\}\ p\ \{\leftarrow p'\}\ (\underline{\mathrm{spec}}\{f'\}{:}tf\ \{\underline{\mathrm{hyp}}\ h{:}th\})\ \{Q\} \\ \qquad\qquad = \underline{\mathrm{spec}}\ f\ \{\leftarrow f''\}{:}tf\ \underline{\mathrm{in}}\ \{P[f,f'\leftarrow f'',f]\}\ \mathrm{body}\ \{Q[f,f'\leftarrow f'',f]\}\ \underline{\mathrm{corp}} \\ \qquad \underline{\mathrm{in}}\ \{R[p\leftarrow p']\}\ \mathrm{stmnt}\ \{S[p\leftarrow p']\}\ \underline{\mathrm{ni}}\ \{S\}\ \ldots \end{array}$$

Of course, {←p'} and {←f"} could be omitted if p and f do not exist in the environment of the block.

2.4. Conclusion

It has been shown that scope rules may be nicely formalized in an axiomatic definition satisfying the criteria c1, c2, c3. Thus one has a measure to judge the mental complexity of the various program constructs (in this section: "syntax restrictions"). In particular it has appeared that unrestricted globality gives rise to rather complicated substitutions and also necessitates a change in the syntax, whereas restricted globality yields no problems at all. Moreover it seems that no further simplification may be gained by further restricting *globality*! This sheds another light on "Global Variables Considered Harmful", Wulf & Shaw (1973).A sequential list of declarations has appeared to be a construct that should be distinguished from a nest of blocks. The disallowance of inner-blocks together with some reasonable syntax assumptions really simplify the semantics.

The more specific deletions from Env, viz -[var v] and -[proc p] should be used when (and allow that) in the same context a name may denote both a var and a proc. This feature for instance should be used to formalize functions where the fun identifier occurring in the body also denotes a var. The next section shows another application.

In general we would like to take all *static* properties into the environment instead of the pre- and postconditions. This might cause some troubles when the ranges are not properly nested, as for instance the <active scope> and <passive scope> of variables in Dijkstra (1976).

3. THE FORMAL TREATMENT OF AN ESCAPE CONSTRUCT

In Bron et al (1976) we have defined the so called escape mechanism for dealing with abnormal termination and abortion of program parts in a highly structured way. The escapes can be used to react to user defined events as well as to system generated error occurrences. They are efficiently implementable and in harmony with the block concept. A fuller treatment - except for the *formal* semantics - can be found in Bron et al (1976).

In this section we give the axiomatic definitions of the construct as originally designed, but with a slightly modified syntax. It will appear that at least criterion c3 has not been satisfied. An attempt to force the satisfaction of c3 will lead to an improvement.

3.1. The original escape construct

Here is a (rather operational) definition of the escapes.
(a) The block new esc e(val:ty) in stmnt ni declares an *escape variable* e.

As usual the declaration binds the name e in stmnt (and makes the programmer aware of the fact that abortions may take place within stmnt). In the scope of the declarations there may occur *calls* and several *definitions* of escape procedures e.
(b) The *block* def escproc e = val y in body corp in stmnt ni defines an escape procedure e. The type of the formal value parameters y is fixed by the declaration of escape variable e. The semantics of an escape procedure differs from normal procedures in that termination of the body causes the block to which the definition is local to be terminated (thus causing *abortion* of stmnt and *abnormal termination* of the block).
(c) Escape procedure calls read the same as procedure calls. Semantically they differ from the usual procedure calls in that their execution causes the "dynamically active" escape procedure to be executed. In other words, escape procedure definitions have dynamic in stead of static scope. (Hence var parameters have been disallowed).
(d) By default the escape variable declaration also means the definition of escproc e = val y in abort corp. Abort is a standard escape variable for which it is assumed that a suitable escape procedure has been defined surrounding the user program.
This completes the definition.

Thus the (implicit) insertion of an escape procedure call can not invalidate the textually succeeding assertion (because assertions have to be interpreted as being true whenever control reaches them). If however the programmer wants to continue an aborted computation anyway, then he should provide some surrounding block, having a weak enough postcondition, with an escape procedure definition. Appendix c gives a realistic example. If no escape procedure definition has been provided by the programmer, abort is called; it will abort program execution and may give an error message and a dump of the stack and the like.

3.2. The axiomatization

Apart from the axiomatization of our escape mechanism, this section is also interesting because of the treatment of "jumps out of procedures". In some axiomatizations of goto-like constructs, such jumps have been disallowed, Donahue (1976), whereas in others true enough they have not been disallowed but neither have procedures been axiomatized (in a way satisfying c2), Kowaltowski (1977). In still others there arise scope problems which have not explicitely dealt with, Clint & Hoare (1972).

As an important, first step we define the formats and some abbreviations expressi the semantics of (a) escape variables, (b) procedures and (c) escape procedures.
(a) The semantics of an escape variable is expressed by esc e(val:ty).
(b) The semantics, psem say, of a procedure is expressed by

psem ≡ proc{P}p(spec{f}:tf {hyp h:th $e_i$assem(i=1..n})){Q}, where
$e_i$assem ≡ escproc {E_i}e{(y_i)}.

The assumed semantics $e_i$assem indicate - in terms of the bound names y_i, the fre names f, h and globals of psem - the precondition E_i on which the escape procedu e_i will be called in the body of p and subsequently control will leave the body. Note that y_i is bound in $e_i$assem and that the binding of f and h extends over $e_i$assem. The three restrictions of section 2.2 are still tacitly assumed.
(c) The semantics, esem say, of an escape procedure is expressed by
esem ≡ esproc {E}e{(y' $e_i$assem(i=1..n))},
where $e_i$assem is as above. Rather that expressing a condition *transformation*, lik psem does, esem expresses a particular condition, viz. the (weakest) preconditio upon which its body establishes the block's postcondition. Hence global variable of the same name as the formal parameters, y say, of its definition might be involved, even if they are not used globally in its body! (Thanks to a suitable choice of syntax esem can be written within the program text in a natural way.) is called the precondition of esem.

(c') The semantics of the escape procedure defined by default has been derived from the axiomatization of abort: true is a sufficient precondition because the programs postcondition will not be reached anymore. So
$\text{initsem} \equiv \underline{\text{escproc}}\ \{\text{true}\}e\{(\ )\}$.

In the rules below we do not explain what should be clear from section 2. Because the free variable names of esem are not merely the names of the variables globally used in its body, we will not assume that globality has been restricted.

The rule for escape variable declaration is not surprising, cfr D1,

$$\text{Esc}\quad \frac{\text{Env}[e\leftarrow e'] + \underline{\text{esc}}\ e(\underline{\text{val}}{:}ty) + \text{initesem} \vdash \{R[e\leftarrow e']\}\ \text{stmnt}\ \{S[e\leftarrow e']\}}{\text{Env} \vdash \{R\}\ \underline{\text{new}}\ \underline{\text{esc}}\ e\ (\underline{\text{val}}{:}ty)\ \underline{\text{in}}\ \text{stmnt}\ \underline{\text{ni}}\ \{S\}}$$

where e' not free in Env, e.

The rule for procedure declaration, cfr D1 and P ,

$$\begin{array}{l}(\text{Env}[p\leftarrow p']-[\underline{\text{escproc}}]+\text{psem})[f\leftarrow f''] + \underline{\text{var}}\ f{:}tf + h{:}th + e_i\text{assem}(i=1..n)\\ \qquad\qquad\vdash \{P[f,f'\leftarrow f'',f]\}\ \text{body}\ \{Q[f,f'\leftarrow f'',f]\}\end{array}$$

$$\text{P4}\quad \frac{\text{Env}[p\leftarrow p'] + \text{psem} \vdash \{R[p\leftarrow p']\}\ \text{stmnt}\ \{S[p\leftarrow p']\}}{\text{Env} \vdash \{R\}\ \underline{\text{new}}\ \underline{\text{proc}}\ p\ (\underline{\text{spec}}{:}tf) = \underline{\text{spec}}\ f{:}tf\ \underline{\text{in}}\ \text{body}\ \underline{\text{corp}}\ \underline{\text{in}}\ \text{stmnt}\ \underline{\text{ni}}\ \{S\}}$$

where p' not free in Env,p
and h, f" distinct and not free in Env,p,p',f.

Thus all escapes globally used in body give rise to explicit assumptions $e_i$assem in psem: each escproc in Env has been deleted in the environment for body!
The rule for escape procedure definition

$$(\text{Env}-[\underline{\text{escproc}}])[y\leftarrow y'']+\underline{\text{var}}\ y{:}ty+e_i\text{assem}(i=1...n)\vdash\{E[y,y'\leftarrow y'',y]\}\text{body}\{S[y,y']\}$$

$$\text{E1}\quad \frac{\text{Env}-[\underline{\text{escproc}}\ e] + \text{esem} \vdash \{R\}\ \text{stmnt}\ \{S\}}{\text{Env} \vdash \{R\}\ \underline{\text{def}}\ \underline{\text{escproc}}\ e = \underline{\text{val}}\ y\ \underline{\text{in}}\ \text{body}\ \underline{\text{corp}}\ \underline{\text{in}}\ \text{stmnt}\ \underline{\text{ni}}\ \{S\}}$$

where y" not free in Env,y.

Thus the justification of esem essentially requires to show that its body establishes the block's postcondition: after termination of body control will goto the end of the block. Note the resemblance with and deviations from D1 and P4. (The following justification of esem does not affect the semantics: replace (Env-[escproc]) by (Env-[escproc] + esem)).
The rules for escape procedure and procedure call. Assume without loss of generality that the bound names y_i of $e_i$assem will not occur free in I,a.

$$\text{EC}_1\quad \frac{\text{Env} + \text{esem} \vdash \underline{\text{escproc}}\ \{E_i[f\leftarrow a]\wedge I\}e_i\{(y_i)\}\ (i=1..n)}{\text{Env} + \text{esem} \vdash \{E[f\leftarrow a]\wedge I\}\ \ e(a)\ \{\text{false}\}}$$

where no actual (global!) var-parameter occurs free in I and

$$\text{PC}_2\quad \frac{\text{Env} + \text{psem} \vdash \underline{\text{escproc}}\ \{\exists h{:}th.E_i[f\leftarrow a]\wedge I\}e\{(\underline{v}_i)\}\ (i=1..n)}{\text{Env} + \text{psem} \vdash \{\exists h{:}th.P[f\leftarrow a]\wedge I\}\ \ p(a)\ \{\exists h{:}th.Q[f\leftarrow a]\wedge I\}}$$

where no actual var-parameter occurs free in I.

So the only difference with rule PC1 is the addition of the premise: it requires to verify in the current environment the actual assumptions, formally made in the justification of esem and psem. The assumptions have been weakened by the addition of the invariant I to their preconditions.

Finally we give two rules, called Semantics verification, enabling us to deduce the premise of EC1 and PC2. The first one is a trivial axiom:

S1: $\text{Env} \vdash e_i\text{sem}(i=1..n)$, provided $e_i\text{assem}(i=1..n) \in \text{Env}$.

The second one is also easy to *formulate:*

let Env contain $\underline{\text{esc}}\ e_i(\underline{\text{val}}{:}ty_i)$ and also

$\underline{\text{escproc}}\ \{E_i\}e_i\{(y_i\ \underline{\text{escproc}}\ \{E_{ij}\}e_{ij}\{(y_{ij})\}(j=1..n_i)\quad)\}(i=1..n)$

and assume without loss of generality that y_{ij} will not occur free in I_i,

then,

$$\text{S2}\quad \frac{\begin{array}{l} Env \vdash \forall y_i{:}ty_i.\tilde{E}_i \supset E_i \quad ,(i=1..n) \qquad\qquad *) \\ Env \vdash \forall y_i{:}ty_i.\tilde{E}_i \supset \tilde{I}_i \quad ,(i=1..n) \\ Env + \underline{\text{escproc}}\ \{\tilde{E}_i\}e_i\{(y_i)\}(i=1..n) \vdash \underline{\text{escproc}}\ \{E_{ij} \wedge \tilde{I}_i\}e_{ij}\{(y_{ij})\}(j=1..n_i,i=1.. \end{array}}{Env \vdash \underline{\text{escproc}}\ \{\tilde{E}_i\}e_i\{(y_i)\}(i=1..n)}$$

where no variable globally used by e_i occurs free in $\tilde{I}_i$.

Thus in order to verify some "weakened $e_i$assem" one has to prove that its precondition $\tilde{E}_i$ implies the one of $e_i$sem already present in Env and one has to verify suitably weakened versions of the assumptions on $e_{ij}(j=1..n_i)$ made by e_i. Because of recursive activation the environment component in the third premise has been extended by what is to be verified. Note that the third premise either invokes S1 or invokes - recursively - S2 again; the number of invocations however is bounded by the cardinality of the set of mutually dependent escape procedure semantics in Env. Appendix B contains a sample proof fully exploiting the recursive character of S2.

Any attempt to associate a unique condition with each escape variable will result in incompleteness, as shown by the following example:

```
new esc e( ) in def escproc e = ... {S1} corp in v:=0;{v=0} e ( ) ni {S1} ...
                def escproc e = ... {S2} corp in v:≠0;{v≠0} e ( ) ni {S2} ... ni
```

This should be compared with usual variables; also with them it is impossible to associate a unique value.

3.3. Simplifying the semantics

Although the definition in terms of normal procedures, 3.1, looks quite simple, it might be argued that the escape mechanism is too complicated: the rules of semantics verification, S1 and S2, are rather exceptional. Not only because their right hand side is the semantics of a name in stead of the behaviour of a program text, but also because they do not satisfy criterion c3: there is a lot of reasoning prescribed for each (implicit or explicit) call of an escape procedure, see again appendix B.

Thus it seems worthwhile to reconsider the design of the escape mechanism, without weakening its power too much. Clearly the inattactive part of rule S2 is its third premise, necessitated by the appearance of $\underline{\text{escproc}}$ assumptions in the semantics of an escape procedure. It reflects that at the site of call, and not known from their definitions, escape procedures may turn out to be *mutually* activating, even if no simultaneous (recursive) definitions have been allowed!

Once the $\underline{\text{escproc}}$ assumptions have been eliminated from the escape procedure semantics esem, the third premise of S2 is no longer needed and we may merge the remainder of it together with S1 into the rules for procedure and escape procedure call:

let Env contain psem, $\underline{\text{esc}}\ e_i(\underline{\text{val}}{:}ty_i)$ and $\underline{\text{escproc}}\ \{\tilde{E}_i\}e_i\{(y_i)\}$

and assume that y_i will not occur free in I and a, then

$$\text{PC3}\quad \frac{Env \vdash \forall y_i{:}ty_i.\ (\exists h{:}th.E_i[f \leftarrow a] \wedge I) \supset \tilde{E}_i\,(i=1..n)}{Env \vdash \{\exists h{:}th.P[f \leftarrow a] \wedge I\}\ p(a)\ \{\exists h{:}th.Q[f \leftarrow a] \wedge I\}}$$

where no actual var parameter occurs free in I

*) equivalently: $Env[y_i \leftarrow y_i'] + y_i{:}ty_i \vdash \tilde{E}_i \supset E_i\,(i=1..n)$ where y_i' not free in Env,y_i.

and for escape procedure call a simple axiom suffices,

$$\text{EP2: } Env + \underline{esc}\ e\ (\underline{val}{:}ty) + \underline{escproc}\ \{E\}e\{(y)\} \vdash \{E[y\leftarrow a]\}e(a)\{false\}.$$

It only remains to eliminate the escproc assumptions from esem. These may be eliminated either by undesirable "syntax restrictions" (like: escapes may not be used globally in escape procedures) or by the following change in the semantics, affecting rule E1 for escape procedure definition: escape procedure bodies, unlike procedure bodies, should be evaluated with a binding for their global escapes fixed in the environment Env in which the definition is considered. This leads to the following rule.

Let esem ≡ escproc {E}e{(y')} - without the e_i assem -.

$$\text{E2}\quad \frac{\begin{array}{l}(Env - [\underline{escproc}\ e] + esem)[y\leftarrow y''] + \underline{var}\ y{:}ty \vdash \{E[y,y'\leftarrow y'',y]\}body\{S[y,y'\leftarrow y'',y]\}\\ Env - [\underline{escproc}\ e] + esem \vdash \{R\}\ stmnt\ \{S\}\end{array}}{Env \vdash \{R\}\ \underline{def}\ \underline{escproc}\ e = \underline{val}\ y\ \underline{in}\ body\ \underline{corp}\ \underline{in}\ stmnt\ \underline{ni}\ \{S\}}$$

where y" not free in Env,y.

The binding might be called "static relative to" the dynamic instance of the procedure body to wich the definition is local, because Env may contain escproc semantics *assumed, but not statically known,* in the surrounding procedure body. Appendic D summarizes the rules. Appendix C contains a realistic example of the use of escapes, proved correct according to the new rules. (the program is also correct according to the original rules).

3.3. Conclusion

Based on the axiomatization of the escape mechanism it was felt that at least some features were too complicated. The attempt to simplify the semantics has been directed by a consideration of the formal rules. Very strikingly, it was only afterwards that we recognized that the formal simplification has resulted in an intuitive improvement as well: now the activation of an escape procedure really escapes the block to which it is local. Thus the above development of the escape mechanism is quite a nice demonstration of the influence of Hoare-like semantics on the design of programming language concepts, as advocated by Gries (1976) and others.

It might be interesting to note that the complexity in the original mechanism has resulted from an attempt to keep things simple: an exceptional kind of procedure, escape procedure,was our starting point and in order to have as few exceptions as possible escape procedure bodies were treated as much as possible like procedure bodies! This decision has now been changed. For the same reason the definition of an escape procedure was originally termed a declaration and we not only expected but also aimed at a similarity in the formalization of procedures and escape procedures. The axiomatizations however clearly show that we had better distinguish between the two concepts and their syntactic appearances.

ACKNOWLEDGEMENT.

I am grateful to Coen Bron for many stimulating discussions.

4. REFERENCES

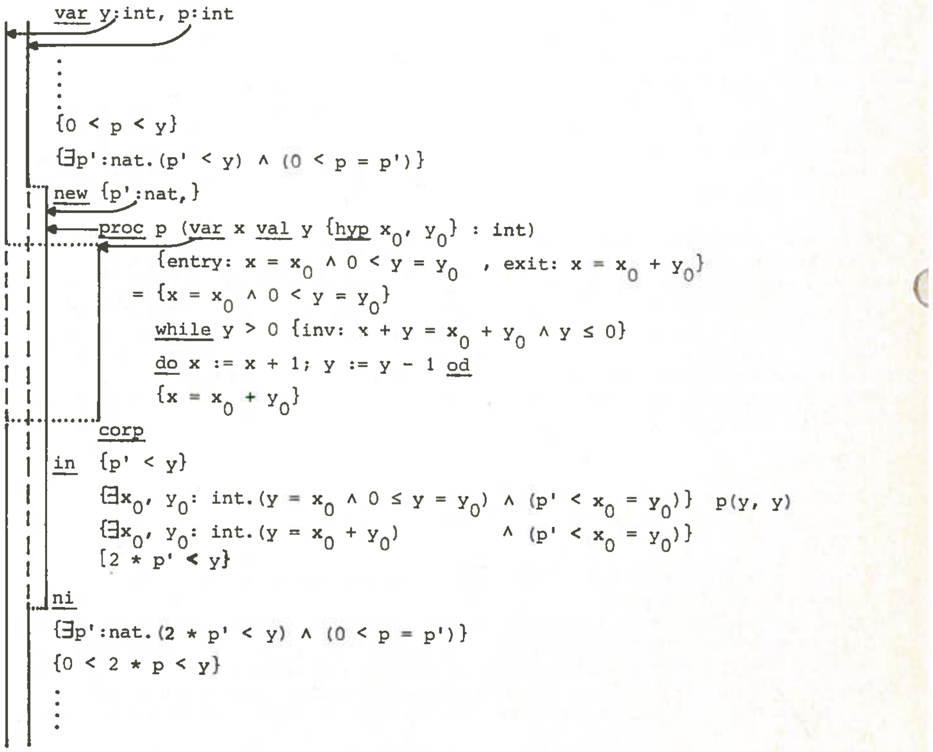
1. Apt, K.R. & de Bakker, J.W., (1977): Semantics and proof theory of Pascal procedures, to appear.
2. Bron, C. & Fokkinga, M.M. & de Haas, A.C.M. (1976): A proposal for dealing with abnormal termination of programs. TW-memorandum 150, T.H.T. Enschede, Netherlands, november 1976. Also to appear elsewhere.
3. Clarke jr., E.M. (1976): Programming language constructs for which it is impossible to obtain good Hoare-like axioms, TR-76-287 Cornell University Ithaca.
4. Dijkstra, E.W. (1976): A discipline of programming, Prentice-Hall Inc.,

Englewood Cliffs, New Yersey.

5. Donahue, J.E. (1976): Complementary definitions of programming language semantics, (thesis) Lect. Notes in Comp. Science 42 (1976) (Springer Verlag).
6. Gries, D. (1976): Some comments on programming language design. In Programmier Sprachen, 4. Fachtagung der GI, Informatik Fachberichte 1 (1976) 135-252 (Springer-Verlag).
7. Hoare, C.A.R., (1971): Procedures and parameters: an axiomatic approach. In Lect. Notes in Math. 188 (1971) 102-116, (springer-Verlag).
8. Hoare, C.A.R. & Wirth, N. (1973): An axiomatic definition of the programming language Pascal, Acta Inf. 2 (1973) 335-366.
9. Knuth, D. (1968): Semantics of context-free languages, Math. Systems Theory 2 (1968) 127-145.
10. Kowaltowski, T., (1977): Axiomatic approach to side effects and jeneral jumps, Acta Inf. 7 (1977)4, 357-360.
11. Ligler, G.T. (1975): Proof rules, mathematical semantics and language design, Ph. D. Thesis, Oxford University (1975).
12. Tennent, R.D. (1976): The denotational semantics of programming languages, CACM 19 (1976 8, 437-453.
13. Wulf, W. & Shaw, M. (1973): Global variables considered harmful, SIGPLAN Notice 8 (1973) 2, 28-34.

Appendix A. A proof fully exploiting P1 anf PC1

For each textpart the environment consists of the semantic descriptions connected to some vertical line in front of the textpart; it is just the usual notion of scope which has been visualized.



```
var y:int, p:int
    :
    :
    {$0 < p < y$}
    {$\exists p'$:nat.($p' < y$) $\wedge$ ($0 < p = p'$)}
new {p':nat,}
    proc p (var x val y {hyp $x_0$, $y_0$} : int)
            {entry: $x = x_0 \wedge 0 < y = y_0$ , exit: $x = x_0 + y_0$}
          = {$x = x_0 \wedge 0 < y = y_0$}
            while y > 0 {inv: $x + y = x_0 + y_0 \wedge y \leq 0$}
            do x := x + 1; y := y - 1 od
            {$x = x_0 + y_0$}
    corp
in  {$p' < y$}
    {$\exists x_0$, $y_0$: int.($y = x_0 \wedge 0 \leq y = y_0$) $\wedge$ ($p' < x_0 = y_0$)}  p(y, y)
    {$\exists x_0$, $y_0$: int.($y = x_0 + y_0$)          $\wedge$ ($p' < x_0 = y_0$)}
    [$2 * p' < y$}
ni
{$\exists p'$:nat.($2 * p' < y$) $\wedge$ ($0 < p = p'$)}
{$0 < 2 * p < y$}
    :
    :
```

Appendix B. A proof essentially involving S2.

The environment has been indicated in the same way as in appendix A.

```
esc one ( ), esc two ( ), var w


def escproc {true} one {(escproc {w ≥ 0} two {( )} )}
   = if w < 3 then w :=1{w = 1}
              else w := w - 3; {w ≥ 0}two ( ) {false}{w = 1}          ...(a)
     fi {w = 1 - the block's postcondition}
in def escproc {true} two {(escproc {w ≥ 3} one {( )} )}
      = if w < 2 then w := 2 {w = 2}
                 else w := w + 1; {w ≥ 3} one ( ) {false}{w = 2}      ...(b)
        fi {w = 2 - implies the block's postcondition: w = 1 ∨ w = 2}
   in {true} two ( ) {false}{w = 2}                                  ...(c)
Env
   ni {w = 2}{w = 1 ∨ w = 2}
ni {w = 1 ∨ w = 2}
```

Proof of calls on line (a) and (b) immediate by S1! We prove line (c) below. Because in each application of S2 (n = 1 and n_1 = 1 and) $\tilde{I}_1$ = true, we omit the second premise of S2.

To be proven Env ⊢ {true} two ( ) {false} , from (1) by EC1.

Note that escproc {true} two {(escproc {w ≥ 0} one {( )})} ∊ Env,

(1) Env ⊢ escproc {w ≥ 0} one{( )} , from (2,3) by S2.

Note that escproc {true} one {(escproc {w ≥ 3} two {( )})} ∊ Env,

(2) Env ⊢ w ≥ 0 ⊃ true , assumed,

(3) Env + escproc {w ≥ 0} one {( )}⊢ escproc {w ≥ 3} two {( )} , from (4,5) by S2.

Let Env + escproc {w ≥ 0} one {( )} be denoted by Env',

note that escproc {true} two{(escproc {w ≥ 0} one {( )})} ∊ Env'

(4) Env'⊢ w ≥ 3 ⊃ true , assumed,

(5) Env' + escproc {w ≥ 0} two {( )}⊢ escproc {w ≥ 0} one{( )} , by axiom S1.

End of proof.

Appendix C. Realistic use of escapes, proved correct according to section 3.3.

The environment has not explicitely been indicated.

Assume a (standard)environment containing:

```
esc overflow( ),
proc recip (var x val y {hyp $y_0$ escproc {$y_0$ too small} overflow {( )}})
     {entry: y = $y_0$  , exit: x = 1/$y_0$}.
```

A program dealing with matrix inversion may read:

```
new esc sing( );
    proc invert (var X val M {hyp $M_0$ escproc {$M_0$ singular} sing {( )}})
         {entry: M = $M_0$  , exit: X = $M_0^{-1}$}
       = def escproc {$M_0$ singular} overflow {( )}
             = {$M_0$ singular} sing ( ) {false}{X = $M_0^{-1}$} corp
         in (* standard inverting, using recip for reciprocal: *)
             ....
             {(a too small ⊃ $M_0$ singular) ∧ ....}
             {∃$y_0$.(a = $y_0$) ∧ ((a = $y_0$) ∧ ($y_0$ too small ⊃ $M_0$ singular) ∧ ...)}
              recip (r, a {∃$y_0$.($y_0$ too small) ∧ (... ∧ ($y_0$ too small ⊃ $M_0$ sing)...)
                             implies $M_0$ singular})
              {∃$y_0$.(r = 1/$y_0$) ∧ ((a = $y_0$) ∧ ........)}
              {r = 1/a ∧ ......}
              ......
         ni {X = $M_0^{-1}$}
    corp
in  .
    .
    .
    .
    (* inner part of a loop in which matrices are computed and inverses printed: *)
    def escproc {i-th matrix singular} sing {( )}
        = {i-th matrix singular} write (i, "singular") {blocks postcond} corp
    in (* overflow caused by compute aborts at least this block! *)
        compute (A);
       {∃$M_0$.(A = $M_0$) ∧ ($M_0$ is i-th matrix)}
       invert (A, A {$M_0$ singular ∧ $M_0$ is i-th matrix ⊃ i-th matrix singular});
       {∃$M_0$.(A = $M_0^{-1}$) ∧ ($M_0$ is i-th matrix)}
       {A is i-th inverse}
        write (A)
    ni {i-th inverse or singularity message printed}
    .
    .
    .
    .
ni.
```

Appendix D. Summary of the rules for the escapes.

Consider the following abbreviations.

$\mathrm{psem} \equiv \underline{\mathrm{proc}}\ \{P\}\ p(\underline{\mathrm{spec}}\ \{f'\}:tf\ \{\underline{\mathrm{hyp}}\ h:th\ e_i\mathrm{assem}(i=1..n)\})\ \{Q\}$

$e_i\mathrm{assem} \equiv \underline{\mathrm{escproc}}\ \{E_i\}\ e_i\{(y_i)\}$

where f', h, e_i distinct and the formal val parameters not free in Q and the variables used globally by p free in psem,

$\mathrm{esem} \equiv \underline{\mathrm{escproc}}\ \{E\}\ e\ \{(y')\}$

$\mathrm{initesem} \equiv \underline{\mathrm{escproc}}\ \{\mathrm{true}\}\ e\ \{(y')\}.$

Escape variable declaration:

$$\frac{\mathrm{Env}[e\leftarrow e'] + \underline{\mathrm{esc}}\ e\ (\underline{\mathrm{val}}:ty) + \mathrm{initesem} \vdash \{R[e\leftarrow e']\}\ \mathrm{stmnt}\ \{S[e\leftarrow e']\}}{\mathrm{Env} \vdash \{R\}\ \underline{\mathrm{new}}\ \underline{\mathrm{esc}}\ e\ (\underline{\mathrm{val}}:ty)\ \underline{\mathrm{in}}\ \mathrm{stmnt}\ \underline{\mathrm{ni}}\ \{S\}}$$

where e' not free in Env,e.

Procedure declaration:

$$\frac{\begin{array}{l}(\mathrm{Env}[p\leftarrow p']-[\underline{\mathrm{escproc}}] + \mathrm{psem})[f\leftarrow f''] + \underline{\mathrm{var}}\ f:tf + h:th + e_i\mathrm{assem}(i=1..n)\\ \qquad\qquad \vdash \{P[f,f'\leftarrow f'',f]\}\ \mathrm{body}\ \{Q[f,f'\leftarrow f'',f]\}\\ \mathrm{Env}[p\leftarrow p'] + \mathrm{psem} \vdash \{R[p\leftarrow p']\}\mathrm{stmnt}\ \{S[p\leftarrow p']\}\end{array}}{\mathrm{Env} \vdash \{R\}\ \underline{\mathrm{new}}\ \underline{\mathrm{proc}}\ p\ (\underline{\mathrm{spec}}:tf) = \underline{\mathrm{spec}}\ f:tf\ \underline{\mathrm{in}}\ \mathrm{body}\ \underline{\mathrm{corp}}\ \underline{\mathrm{in}}\ \mathrm{stmnt}\ \underline{\mathrm{ni}}\ \{S\}}$$

where p' not free in Env,p

and h,f" distinct and not free in Env,p,p',f.

Escape procedure definition:

$$\frac{\begin{array}{l}(\mathrm{Env} - [\underline{\mathrm{escproc}}\ e] + \mathrm{esem})[y\leftarrow y''] + \underline{\mathrm{var}}\ y:ty \vdash \{E[y,y'\leftarrow y'',y]\}\ \mathrm{body}\ \{S[y,y'\leftarrow y'',y]\}\\ \mathrm{Env} - [\underline{\mathrm{escproc}}\ e] + \mathrm{esem} \vdash \{R\}\ \mathrm{stmnt}\ \{S\}\end{array}}{\mathrm{Env} \vdash \{R\}\ \underline{\mathrm{def}}\ \underline{\mathrm{escproc}}\ e = \underline{\mathrm{val}}\ y\ \underline{\mathrm{in}}\ \mathrm{body}\ \underline{\mathrm{corp}}\ \underline{\mathrm{in}}\ \mathrm{stmnt}\ \underline{\mathrm{ni}}\ \{S\}}$$

where y" not free in Env,y.

Procedure call:

Let psem, $\underline{\mathrm{esc}}\ e_i(\underline{\mathrm{val}}:ty_i)$, $\underline{\mathrm{escproc}}\ \{\tilde{E}_i\}\ e_i\{(y_i)\} \in \mathrm{Env}$

and assume y_i not free in I and a, then

$$\frac{\mathrm{Env} \vdash \forall y_i\,ty_i.(\exists h:th.E_i[f\leftarrow a] \wedge I) \supset \tilde{E}_i(i=1..n)}{\mathrm{Env} \vdash \{\exists h:th.P[f\leftarrow a] \wedge I\}\ p(a)\ \{\exists h:th.Q[f\leftarrow a] \wedge I\}}$$

provided no actual var parameter free in I.

Escape procedure call:

$\mathrm{Env} + \underline{\mathrm{esc}}\ e(\underline{\mathrm{val}}:ty) + \underline{\mathrm{escproc}}\ \{E\}\ e\ \{(y)\} \vdash \{E[y\leftarrow a]\}\ e(a)\ \{\mathrm{false}\}.$