TECHNISCHE HOGESCHOOL TWENTE

MEMORANDUM NR. 159

ON THE USE OF CONTINUATIONS IN MATHEMATICAL SEMANTICS

M.M. FOKKINGA

JANUARY 1977

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

ONDERAFDELING DER TOEGEPASTE WISKUNDE

Contents

0. <u>Abstract</u>.

With respect to jumps it is shown how to structure the definition of the mathematical semantics of a programming language with the following two related objectives.
First, the definition should not be easily adapted such that it describes less restrictive jumps. Second, from the definitional text itself the possible flows of control should be fully determined. These objectives are in the spirit of structured programming (now applied to semantics description) in that as much as possible should be clear by local inspection of the text.

<u>Keywords</u>.

Continuation, mathematical semantics, jump, escape.

1. <u>Introduction</u>.

The intention of Mathematical Semantics, also called Denotational Semantics, is to describe in well defined mathematics the semantics of a programming language. The mathematical language thus is a purely applicative language, that is, the only concepts used are (function) definition and (function) application and there is no concept of control (which causes an untimely abortion of the evaluation of an expression) nor the concept of variable (which can change of value). By now there is a well known technique of describing the flow of control, as caused by <u>goto</u>'s, in a purely applicative language, viz. by means of continuations, (Strachey & Wadworth [1]). Roughly, continuations represent the rest of the computation to be performed after terminating the program text under consideration.
In this paper I show how to design and to use the continuations in order to get a definition *precisely suited* for the kind of jumps under consideration. Here, "precisely suited" means that the definition can not be easily changed (that is, without affecting the structure) so that it also describes the

semantics of less restrictive jumps. This has the advantage that the (mental ?) complexity of some less restrictive jumps can be measured by the amount of change made in the definition of a more restrictive jump to allow for the less restrictive one. Additionally, a "precisely suited" definition should allow to infer as many as possible properties (of the flow of control) from the definitional text itself without forcing to prove intricated lemma's. This is fully in the spirit of structured programming, where among others the intention is that the behaviour of the program in execution should be as clear as possible from the program's text itself. (And this may be achieved by modularity, stepwise refinement and so on). Thus the importance of this paper lies not in the field of new theoretical results but instead in the field of methodology of designing mathematical semantics.

This investigation has been inspired by the thesis of Donahue [2] ; he has given a mathematical semantics of a subset of PASCAL where no jumps are allowed out of procedures and he claimed the definition be precisely suited, in the first sense described above, for such jumps. (It will be clear from the sequel that he isn't quite right). In addition, the characterization of the state of a program in execution by the values of precisely the currently visible variables is taken from his work, and this has strong consequences on the role of continuations.

2. The Programming Language and the Mathematical Language.

In order not to be confused by irrelevant details I consider a rather simple Programming Language. It contains the conditional and repetitive statement, procedure declaration and procedure call, and jumps of the following kind. A block may contain an escape declaration to which a transfer of control is made upon execution of an <u>escape</u> statement (usually called the <u>goto</u> statement). The defining statement in the escape declaration is executed and then control <u>goes</u> <u>to</u> the end of the block.

```
Block ::= new decl in stmnt ni,
Stmnt ::= id := expr | id(id* : expr*) | esc id | stmnt ; stmnt |
          if expr then stmnt else stmnt fi |
          while expr do stmnt od | block,
Decl  ::= var id := expr |
          proc id(id* : id*) = stmnt |
          esc id = stmnt,
```

According to whether the declaration of a block is a <u>var</u>, <u>proc</u>, <u>esc</u> declaration it is called a <u>var</u>-, <u>proc</u>- or <u>esc</u>-block.
I will consider language variants in which escapes are allowed (a) not at all, (b) out of a single block (c) out of a multiple escape block, (d) out of any block, but for (a)-(d): not out of procedures, and (e) out of any block even out of procedures.
Neither the syntax nor the semantics of expressions is considered in detail. The only assumption made is that their evaluation has no side effect (that is, does not change the value of any variable).
Further, no global variables are allowed and the colon in the parameter list of the procedure separates the <u>var</u> parameters from the <u>val</u> ones (cf. Hoare [3]), in the sense of the PASCAL <u>var</u> and <u>value</u> parameters. Both the lists of formal parameters and the list of actual <u>var</u> arguments must be lists of distinct names. Also, both the procedures and the escapes are recursive in the sense that the range of the name being defined also extends over the defining stmnt.
One final remark is in order. Because no global variables are allowed the "state" at any point of the computation is fully determined by the values of the *currently visible* variables. The definition of the semantics will make this explicit and at the same time it will be clear that no information is lost upon block entry and exit and upon procedurebody entry and exit.

The mathematical language only contains (function) definition and (function) application. All the domains are complete lattices and justified by the work of Scott [4,5] extensive use is made of recursive definitions. It must be stressed however, that my mathematical language is call-by-value rather than call-by-name. That is, whenever in an application some of the argument values are $\perp$ then the final result is $\perp$ and otherwise, whenever some of the argument values are $\top$ then the final result is $\top$. The only and fundamental exception is the <u>if</u> <u>then</u> <u>else</u>, which is call-by-value on its first argument only. Following the suggestions of Reynolds [6], I could transform the definitions so that they become independent of the order of evaluation but this would complicate the formulae. Frequently I use the form "expr1 <u>where</u> x = expr2" for readability to abbreviate (λx.expr1) (Yλx.expr2) where Y is the least fixed point operator.

I write (f a b c) instead of f(a,b,c), and $f : A \times B \times C \to D$ is identified with $f : A \to B \to C \to D$, so that $(f\ a\ b) : C \to D$ makes sense as well as $(f\ a) : B \to C \to D$. I omit outermost parentheses. As usual, for $f : A \to B$, $a : A$ and $b : B$, the "substitution" $f[a \leftarrow b]$ abbreviates (subst f a b) <u>where</u>

$$\text{subst } f\ a\ b = \lambda x : A.\ \underline{\text{if}}\ x = a\ \underline{\text{then}}\ b\ \underline{\text{else}}\ f\ x.$$

The postfix substitution operator has precedence over function application so that $(a\ b[c \leftarrow d])$ means $(a\ (b[c \leftarrow d]))$.

The + denotes *disjoint union*. If $U = \underline{d1} : D1 + \ldots + \underline{dn} : Dn$ then U consists precisely of the elements $\neq \top$, $\neq \bot$ of the Di tagged with <u>di</u> and another new $\top$ and $\bot$. The symbol <u>di</u> is used for three purposes : *injection* $\underline{di} : Di \to U$, *projection* $\underline{di} : U \to Di$ and *inspection* $\underline{di} : U \to \text{Bool}\ \{= \bot, tt, ff, \top\}$. An application of the latter one is written as $U : \underline{di}$ so that the context determines which one is intended.

The following axioms define the <u>di</u>.

(a) on $\bot$ ($\top$) they yield $\bot$ ($\top$)

(b) for $d : Di$ and $\bot \neq d \neq \top$, $(\underline{di}\ d) : \underline{di} = tt \wedge \underline{di}\ \underline{di}\ d = d$

(c) for $u : U$ and $\bot \neq u \neq \top$,

either $u : \underline{di} = tt \wedge \underline{di}\ \underline{di}\ u = u \wedge u : \underline{dj} = ff$ for $j \neq i$

or $u : \underline{di} = ff \wedge \underline{di}\ u = \top$.

The top element $\top$ is used as the erroneous result. In particular, in a disjoint union the projection of an element into a summand from which the element did not originate, yields the erroneous value $\top$. Because of the call by value restriction an error value $\top$ cannot disappear except for replacement by $\bot$.

3. <u>The Simplest Language : No Jumps at all.</u>

The state (of the programming system) should be a function from the class of Identifiers yielding at each application either a message that the identifier has not been declared as a <u>variable</u> or a message that it is a variable and has a value. So

$$\text{State} = \text{Id} \to (\underline{\text{udef}} : \text{Null} + \underline{\text{val}} : \text{Val})$$

Here, Null is a domain, the elements of which will not be used; it seems not

possible however to have an empty domain, therefore let Null have a single proper element which is denoted by ! Note that if (s id) : udef then val (s id) is the erroneous value $\top$.
Similarly the environment is modelled by
Env = Id → (udef : Null + proc : Proc),
Proc = Val* → State.
I leave the domain Val uninterpreted (think of integers e.g.)

Now the meaning of a statement is a statetransformation depending on the environment in which it is executed.
Ms : Stmnt → Env → State → State,
and similarly, the type of the meaning function for expressions is
Me : Expr → State → Val
Now, I give the clauses of the definition of Ms.
Ms ⟦id := expr⟧ e s = s[id ← val (Me expr s)]
thus the terminal state really is the same as the initial one except that id has a new value,
Ms ⟦st1;st2⟧ e s = Ms st2 e (Ms st1 e s)
the terminal state of st1 is used as the initial one for st2,
Ms ⟦if exp then st1 else st2⟧ e s =
if (Me exp s) then (Ms st1 e s) else (Ms st2 e s),
Ms ⟦while expr do stmnt od⟧ e s =
if (Me expr s) then (Ms ⟦stmnt;while expr do stmnt od⟧ e s) else s,
Ms ⟦new decl-of-id in stmnt ni⟧ e s =
(Ms stmnt newe news) [id ← s id]
where newe = e[id ← ...] and news = s[id ← ...]
and it depends on the kind of declaration what should be placed on the dots. In any case either in e or in s the identifier must be hidden, that is, set to udef! Note also, that whatever the declared identifier is, after terminating the execution of stmnt the identifiers original value - whatever it was - is restored in the terminal state. Now, in case the declaration is var id := expr then the replacement should be e [id ← udef!] and s [id ← val (Me expr s)] and in case of proc id(x:y) = body we have
s [id ← udef!] and e [id ← proc f]
where f = λa:Val*, b:Val*.
(Ms body proce procs) [y ← udef!]
where proce = newe [x,y ← udef!, udef!]
procs = udef [x,y ← val a, val b]*)

*) Of course, the state udef is λid:Id. udef!

Thus it is obvious from the definitional text that upon entry of body only x and y are known in the state and that after terminating the body y is made invisible - because it is intended to be a local variable. Further, the procedure's meaning f is such that when applied to suitable arguments it results in a state in which only x - the formal <u>var</u> parameter - is "known", hence I define (rather loosely):

<u>Ms</u> $⟦$p(a:expr)$⟧$ e s = s[a $\leftarrow$ (<u>proc</u>(e p)) (<u>val</u>(s a)) (<u>Me</u> expr s) x],

thus modelling a procedure call as a simultaneous substitution of new values (computed by p) for the variables a (cfr the PASCAL axiomatization of procedures [7]).

Two remarks are in order.

First the mathematical semantics does not necessarily yield an error value if the program is not syntactically correct. For instance, an assignment to some undeclared variable is well defined and even the occurence of global variables in procedure bodies does not necessarily result in $\top$. You might try to treat syntax restrictions in the mathematical semantics but that yet requires a separate (pre)processing of the program text, because the text of an unreachable conditional branch is not evaluated at all by <u>Ms</u>. Therefore I leave it as a merely syntactical matter. Note however, that it is clear from the definitional text that the initial value of global variables does not contribute to the terminal state of a procedure!

Second, some syntax restriction really is necessary in order that the semantics are well-defined. For instance, a substitution [x $\leftarrow$ val] is ill defined if x is not a list of distinct objects. This applies for instance to both the list of all formal parameters and the list of actual variable arguments. (But if you prefer you may consider the semantics as specifying a nondeterministic result. Or you may make the convention that such denotations of substitutions should be understood as a left-to-right specification. It all depends on what language you want to define).

4. <u>Allowing jumps in the Language</u>.

Now there arises a difficulty in describing the semantics in a purely applicative language, if we stick to the method of specifying the semantics by induction to the syntax rules. For the final state after the execution of a stmnt need not necessarily be the initial one of the textually succeeding stmnt, but it may as well be the terminal one of some escaped block.

The well known techniques of continuations solves this problem in the following way: the type of Ms is now

Ms : Stmnt → Cont → Env → State → State, where

Cont = State → State

Env = Id → (udef : Null + proc : Proc + esc : Esc)

Proc = the domain of the procedures meanings

Esc = Cont {the escape's meanings},

and in the formula "Ms stmnt c e s" the c represents the rest of the computation to be done after properly terminating stmnt, while the escape continuations in e are used to yield the final state if there occurs an escape out of stmnt (see Tennent [8], Strachey [1]).

Thus, whatever kind of escape is allowed, we have the following clauses in the definition of Ms:

Ms⟦st1;st2⟧ c e s = Ms st1 (Ms st2 c e) e s

"just yield the final state of st1 applied in e to s with - when terminated properly - followed by the continuation which first applies st2 and then - when terminated properly - the original continuation c",

Ms⟦if expr then st1 else st2 fi⟧ c e s =

if(Me expr s) then (Ms st1 c e s) else (Ms st2 c e s),

Ms⟦while expr do stmnt od⟧ c e s =

if (Me expr s) then (Ms ⟦stmnt;while expr do stmnt od⟧ c e s) else (c s)

Recall that expressions cannot have side effects and cannot be escaped from.

Also, the meaning of an assignment and escape statement is fixed:

Ms⟦id:=expr⟧ c e s = c s[id ← val (Me expr s)]

"just apply the rest of the computation to the state s in which id is set to the value of expr",

Ms⟦esc id⟧ c e s = (esc (e id)) s

"just discard the normal continuation c and apply the escape continuation to s".

Now, I could give several further clauses to define the meaning of blocks and procedure calls, for the most general kind of jumps you can think of. This is done in the litterature [8,6,9] and I will do it too for my simple programming language in section 8. The restricted jumps then are defined as well, because the syntax restrictions prohibit that programs utilize the full power of an escapes meaning. But then the definitional text is not *precisely suited* for the restricted jumps: without any change it models the unrestricted jump as well and in addition, without taking the syntax restrictions into account it is not possible to infer from the definitional text that certain jumps are not possible.

In the following sections I will present a definition of <u>Ms</u> which really is precisely suited for the kind of jump under consideration. In order to be convincing I need definitions of the notions of transfer of control which I am interested in.

(4.1) <u>Definition</u>. Let stmnt: Stmnt, c : Cont, e : Env and s, s' : State.

1. The *final state* of stmnt with respect to initial c, e and s is
   <u>Ms</u> stmnt c e s,

and here c is called the *normal continuation* and for each id such that (e id) : <u>esc</u>, <u>esc</u>(e id) is called an *escape continuation*.

2. The *null continuation* nullc, which does not contribute anything to the computation is λs : State. s .

3. Stmnt is *abortion free* iff
   $\forall$c, e, s. <u>Ms</u> stmnt c e s = c (<u>Ms</u> stmnt nullc e[<u>esc</u>: $\leftarrow$ <u>udef!</u>] s) *)

Informally it means that control is never transferred out of stmnt by means of an escape statement. This property is the semantical counterpart of the syntax property that stmnt does not contain escape statements to non-local escape identifiers.

4. Stmnt with initial e and s *is aborted* on s', the *abortion state*, resp. *terminates on s'*, the *terminal state*, iff
   $\exists$ id : Id $\forall$f : Id $\rightarrow$ (<u>udef</u> : Null + <u>esc</u> : Cont) $\forall$c : Cont.
   <u>Ms</u> stmnt c ef s = (<u>esc</u>(ef id)) s' , resp. = c s'
   <u>where</u> ef = λid. <u>if</u> (e id) : <u>esc</u> <u>then</u> <u>esc</u> (<u>esc</u>(f id)) <u>else</u> (e id).

Informally s' is the state when control leaves stmnt; in case of abortion control is transferred out of stmnt by means of an escape, and vice versa. Note however, that infinite computation *within* stmnt is termed *termination* with terminal state $\perp$.

Immediate from the definitions there follows

---

*) e[<u>esc</u>: $\leftarrow$ <u>udef!</u>] abbreviates λid. <u>if</u> (e id): <u>esc</u> <u>then</u> <u>udef!</u> <u>else</u> (e id).

4.2. <u>Lemma</u>

a. if s is not $\perp$ or $\top$ then it is either an abortion state or a terminal state but not both (provided Id and Val are non trivial).

b. if s' is terminal then $s' =$ <u>Ms</u> stmnt nullc e s, hence

c. if stmnt with e and s terminates then

$\forall c$: Cont. <u>Ms</u> stmnt c e s = c (<u>Ms</u> stmnt nullc e s)

d. the analogi of <u>b</u> and <u>c</u> in case of abortion. □

It should be noted that implication <u>c</u> cannot be reversed. First, it will appear that infinite looping *within* an escape body *outside* stmnt gives rise to a proper abortion state, though in that case $\forall c$. <u>Ms</u> stmnt c e s = $\perp$ = c (<u>Ms</u> stmnt nullc e s). Second, it will appear that each escape body in the language of section 5 (single escape block escapes) satisfies $\forall c$. <u>Ms</u> escbody c e s = c (<u>Ms</u> escbody nullc e s) for the environments and states on which it ever is executed, but yet it may be aborted (by an escape to itself).

Also immediate from the definitions there follows

4.3 <u>Lemma</u> stmnt terminates for all e and s iff it is abortion free □

Thus I use "is guaranteed to terminate" as "terminates for any e and s" and as "is abortion free". The qualification "is potentially aborted" means "there exist e and s such that it is aborted with abortion state $s' \neq \perp, \top$" or equivalently "is not abortion free".

The following lemma however cannot be proven from the preceding definitions above but requires the semantics to be known.

4.4 <u>Lemma</u>

For any e, s and stmnt there is exactly one state s which is either a terminal state or an abortion one or both. <u>Proof</u> Because <u>Ms</u> will be deterministic there is at most one such state. By induction it will be easy to prove that there is at least one such state.

□

By now it is clear from the definitional text of <u>Ms</u> given so far that an assignment is guaranteed to terminate. This is not clear for statement sequencing, conditionals and repetitions; indeed it will appear that they are potentially aborted.

The notion "precisely suited" is formally treated in section 9.

5. Very Simple Jumps : Single Block Escapes.

In this language variant an escape statement is allowed only if the escape identifier is declared in the least surrounding block. So any block terminates and the blocks are the *smallest* program constructs guaranteed to terminate. Consequently we let Ms and each continuation yield the terminal state of the smallest surrounding block.

<u>Ms</u> : Stmnt → Cont → Env → State {initial} → State {terminal one of the block}

Cont = State {when control leaves the stmnt} → State {terminal one of the block}

Env = Id → (<u>udef</u> : Null + <u>esc</u> : Cont + <u>proc</u> : Val^* → State).

And the remaining clauses of <u>Ms</u> are as follows.

<u>Ms</u> ⟦<u>new</u> decl-of-id <u>in</u> stmnt <u>ni</u>⟧ c e s =

c (<u>Ms</u> stmnt nullc newe news) [id ← s id]

<u>where</u> news = s[id ← ...]

newe = e[<u>esc</u> : ← <u>udef</u>!] [id ← ...]

Thus it is clear from the text that each block is abortion free. Depending on the type of declaration I now specify to what value id is set in the environment e and state s:

decl is <u>var</u> id := expr, then

[id ← <u>udef</u>!] in e and [id ← <u>val</u>(<u>Me</u> expr s)] in s,

decl is <u>proc</u> id(x:y) = body, then

[id ← <u>proc</u> f] in e and [id ← <u>udef</u>!] in s

<u>where</u> f = λa : $\underline{\text{Val}}^*$, b : $\underline{\text{Val}}^*$.

(<u>Ms</u> body nullc proce procs) [y ← <u>udef</u>!]

<u>where</u> proce = newe [<u>esc</u> : ← <u>udef</u>!] [x,y ← <u>udef</u>!, <u>udef</u>!]

procs = udef [x,y ← <u>val</u> a, <u>val</u> b]

thus a procedure body is abortion free,

decl is <u>esc</u> id = body, then

[id ← <u>esc</u> f] in e and [id ← <u>udef</u>!] in s

<u>where</u> f = λs : State. <u>Ms</u> body nullc newe s

(may be it is better to *write*

e[<u>esc</u> : ← <u>udef</u>!] [id ← <u>esc</u> f] instead of newe),

thus the escape body potentially is aborted (by an escape to id itself), but yet for any s : ∀c : Cont.

<u>Ms</u> body c newe s = c (<u>Ms</u> body nullc newe s), that is, control eventually reaches the end of the body.

Finally

$\underline{Ms}$ ⟦p(a:expr)⟧ c e s = c s[a ← ($\underline{proc}$(e p)) ($\underline{val}$(s a)) ($\underline{Me}$ expr s) x]

Thus a procedure call is guaranteed to terminate.

6. Rather Simple Jumps : Multiple Escape Block Escapes.

In this language variant escapes are allowed to abort several blocks at a time, provided each of them is an escape block. Consequently, the smallest program construct containing stmnt and guaranteed to terminate is the outermost escape block which contains stmnt but no non-escape block which also contains stmnt. This block is called the multiple escape block. I let both $\underline{Ms}$ and each continuation yield the value of the multiple escape block:

$\underline{Ms}$: Stmnt → Cont → Env → State → State {terminal one of the multiple esc block}

Cont = State {when control leaves stmnt} → State {terminal of the mult esc block}

Env = Id → ($\underline{udef}$: Null + $\underline{esc}$: Cont + $\underline{proc}$: Val^* → State)

and the remaining clauses of the definition of $\underline{Ms}$ read

$\underline{Ms}$ ⟦$\underline{new}$ decl-of-id $\underline{in}$ stmnt $\underline{ni}$⟧ c e s =

{depending on the type of block}

if the type is $\underline{not}$ $\underline{escape}$ *then*

c ($\underline{Ms}$ stmnt nullc newe news) [id ← s id]

$\underline{where}$ newe = e[$\underline{esc}$: ← $\underline{udef}$!] [id ← ...]

news = s [id ← ...]

and if the type is $\underline{escape}$ *then*

$\underline{Ms}$ stmnt newc newe news

$\underline{where}$ newc = rest id s c

newe = (λid. $\underline{if}$ (e id): $\underline{esc}$ $\underline{then}$ $\underline{esc}$ (rest id s $\underline{esc}$ (e id))

$\underline{else}$ (e id))[id ← ...]

news = s[id ← ....].

By now, it is already clear from the text that each non-escape block terminates and that an escape block is potentially aborted by an escape to some surrounding block. Note further that the responsibility for restoring the original value of id in the final state - after leaving the block - is placed both by the normal and the escape continuations. This is achieved by means of the restore function rest, defined by

rest id s c = λfin:State. c fin[id ← s id].

It should be obvious how to complete the substitutions to id in e and s in order to obtain newe and news:

decl is var id := expr, then
    [id ← udef!] in e and [id ← val (Me expr s)] in s,
decl is proc id (x:y) = body, then
    [id ← proc f] in e and [id ← udef!] in s
    where f = λa:Val*, b:Val*.
        (Ms body nullc proce procs) [y ← udef!]
        where proce = newe [esc : ← udef!] [x,y ← udef!, udef!]
            procs = udef [x,y ← val a, val b],
        thus a procedure body is abortion free,

decl is esc id = body, then
    [id ← esc (rest id s f)] in e and [id ← udef!] in s
    where f = λs:State. Ms body newc newe s.

Finally, the definitional clause for procedure call reads the same as before because any call terminates

Ms ⟦p(a:expr)⟧ c e s = c s[a ← (proc(e p)) (val(s a)) (Me expr s) x]

7. Simple Jumps: Escapes from any Block.

For any stmnt the smallest program construct containing stmnt and guaranteed to terminate is now the procedure body. Thus Ms and both the normal and the escape continuations will yield the terminal state of a procedure body.

Ms : Stmnt → Cont → Env → State {of the surrounding proc body}
Cont = State → State {of the surrounding proc body}
Env = Id → (udef : Null + esc : Cont + proc : Val* → State)
and the definitional clauses read:
Ms ⟦new decl-of-id in stmnt ni⟧ c e s =
    Ms stmnt newc newe news
    where newc = rest id s c
        newe = (λid. if (e id): esc then esc (rest id s esc (e id))
                                else e id   ) [id ← ...]
        news = s[id ← ...]

Thus any block potentially is aborted and each continuation restores id's original value in the state when control leaves the block. Further, the substitutions in e and s in order to obtain newe and news are exactly the same as in the previous section.

Thus it is clear from the text that a procedure body cannot be aborted and hence the clause for procedure call still is the same as before:
<u>Ms</u> ⟦p(a:expr)⟧ c e s = c s[a ← (<u>proc</u>(e p)) (<u>val</u>(s a)) (<u>Me</u> expr s) x].

Again, with respect to the preceding language variant only those definitial clauses have been changed for which there is a change in the semantics. The possible transfers of control which I am interested in (abortion, termination) are clear from the text above.

8. Unrestricted Escapes, even out of Procedures.

The smallest program construct now guaranteed to terminate is the complete program itself. Thus <u>Ms</u> and each continuation must yield the final state of the program:

<u>Ms</u> : Stmnt → Cont → Env → State → State {final one of the program}
Cont = State → State {final one of program}
Env = Id → (<u>udef</u> : Null + <u>esc</u> : Cont + <u>proc</u> : Proc).

By consequence, the domain Proc of procedure meanings has to be changed: a procedure body potentially yields the final state of the program as well. And upon the termination and abortion of a procedure body an updated version of the state at procedure call has to be subject to the continuation. So both the state and the normal continuation have to be passed to the procedures meaning:

Proc = Val* → Cont → State → State {final one of the program}.

Although the interpretation of the delivered state has been changed, all the clauses of <u>Ms</u> remain the same except for procedure call and the procedures meaning f:

```
f = λa:Val*, b:Val*, cc{c at call}: Cont, sc{s at call}:State.
    Ms body procc proce procs
    where procc = λs:State. cc sc[a ← s[y ← udef!] x]
          proce = (λid:Id. if (e id): esc then esc (rest id sc esc(e id))
                                            else (e id) ) [x,y ← udef!, udef!]
          procs = udef [x,y ← val a, val b]
```

And still I model a procedures effect as a multiple assignment to the actual var arguments a, which is performed at the moment when control leaves the body (i.e. a continuation is applied by means of termination or abortion).

Note again that precisely those textparts have been changed which model some changed semantics, and that the former definitional text could not be adapted by some substitutions applied to the environment or state.

9. A formalization of the notion "precisely suited".

First of all, the notion "precisely suited" applies to *definitional texts* of meaning functions Ms and should be understood *relative to* the properties of abortion, termination, and abortion free. Hence the restriction is that Ms must be of type Stmnt→Cont → Env → State → State. The intention is, as stated in the introduction, that as much as possible (of the interesting properties) should be clear by local inspection of the text. Hence I could try to define the notion by

9.1 a definition of Ms is precisely suited if and only if

for each program construct the definitional clause of Ms alone is sufficient to prove that the construct is abortion free, iff it is abortion free.

But then it is rather hard to prove that a particular definition is not precisely suited. In any case, the definitions presented in sections 5 through 8 satisfy the requirement. Closer inspection of those definitional texts however, shows a property which I now do take as the definition.

9.2 Definition

A definition of Ms is precisely suited if and only if

for each program construct the definitional clause of Ms has the form
Ms construct c e s = c(- - -), where on the dots every occurrence of e is postfixed with [esc: ← udef!], iff the construct is abortion free.

Indeed it is now rather easy to prove the only-if part of 9.1 as a lemma.

9.3 Proof. Let stmnt be an abortion free construct and let c, e and s be arbitrary. Then Ms stmnt c e s = {by 9.2}

= c (. . . e[esc: ← udef!]. . .) = {by def of nullc}

= c (nullc(. . . e[esc: ← udef!]. . .)) = {by 9.2}

= c (Ms stmnt nullc e s).

Conversely, if stmnt is not abortion free, then it is of course not possible to prove that it is abortion free. □

Note, however, that the if-part of 9.1 does not hold: place the null continuation in front of each right hand side of the definitional clauses. Thus def 9.2 excludes several texts which intuitively could equally well be termed precisely suited. Yet, def 9.2 is not too restrictive:

9.5 Lemma For each definition of Ms there exists a precisely suited one.
proof Let stmnt be an abortion free construct, then by definition

Ms stmnt c e s = c(Ms stmnt nullc e[esc: ← udef!] s) .... (*).

Now replace in the rhs of the given definitional clause of stmnt the continuation c by nullc and the environment e by e[esc: ← udef!] and place c in front of the right hand side. Then by (*) the resulting expression defines Ms stmnt c e s and it is precisely suited. □

So if an escape out of the repetitive statement is disallowed by additional syntax restrictions, then the precisely suited definitional clause of Ms reads as follows

Ms ⟦while expr do stmnt od⟧ c e s =
c (if (Me expr s)
    Then (Ms ⟦stmnt; while expr do stmnt od⟧ nullc e[esc: ← udef!] s)
    else s).

Definition 9.2 indicates the reason why a precisely suited definition cannot be easily adapted to allow for less restrictive escapes: the structure of the definitional clause of the construct, which by weakening the restrictions on the escape statements no longer is abortion free, has to be changed essentially.

In the above exposition it was tacitly assumed that a program construct corresponds to a syntax clause in the definition of the language. I can strengthen the assertion "the definitions of Ms in sections 5 - 8 are precisely suited" by making separate syntax clauses for procedure bodies and escape bodies. For instance

    block ::= new proc id (x:y) = body in stmnt ni,
    body ::= stmnt

and defining as an alternative to sections 5 and 6

Ms block c e s = c(Ms stmnt nullc newe news) [id ← s id]
where news = s[id ← udef!]
    newe = e[esc: ← udef!] [id ← proc f]

$$\underline{\text{where}}\ f = \lambda a{:}Val^*,\ b{:}Val^*.(\underline{Ms}\ body\ \ nullc\ proce\ procs)\ [y \leftarrow \underline{udef!}]$$
$$\underline{\text{where}}\ proce = newe\ [x,y \leftarrow \underline{udef!},\ \underline{udef!}]$$
$$procs = udef\ [x,y \leftarrow \underline{val}\ a,\ \underline{val}\ b]$$
$$\underline{Ms}\ body\ \ c\ e\ s = c\ (\underline{Ms}\ stmnt\ \ nullc\ e[\underline{esc}{:} \leftarrow \underline{udef!}]\ s).$$

So now procedure bodies formally are abortion free constructs.

10. Conclusion.

I have shown how to use continuations: they should yield the value of the smallest program construct guaranteed to terminate, and in defining expressions they should be shifted as far as possible to the outermost level and escape continuations should be hidden as much as possible. This has the advantage that the definitional text itself clearly shows which program constructs can be aborted and additionally, a weakening of the restrictions on jumps is reflected by a change in the definition which characterizes the weakening. Thus one has a measure to judge the (mental?) complexity of the various kinds of jumps.
Simultaneously another aspect of continuations has been become clear: when the state consists of the value-variable correspondence for precisely the currently visible variables, then restoring the original value of the block's identifier upon exit of the block is a task for the continuations.

Added in proof.

"Continuation removal" is a theorem already stated by Ligler [10]. In my formalism and terminology it means that each program text which happens to be *syntactically* jump free is indeed *abortion free*.

References

[1] Strachey, C. and Wadsworth, C.P.: Continuations, a mathematical semantics for handling full jumps, Tech. Mon. PRG-11 (1974), Oxford Univ. Comp. Lab., Programming Research Group.

[2] Donahue, J.E.: Complementary Definitions of Programming Language Semantics (Thesis), Lect. Notes in Comp. Sc. 42 (May 1976), Springer-Verlag, Berlin.

[3] Hoare, C.A.R.: Procedures and Parameter: an axiomatic approach, in Symp. On Sem. of Algorithmic Lang. (Ed: E. Engeler), Lect. Notes in Math. 188 (1970) 102-116.

[4] Scott, D.: Outline of a Mathematical Theory of Computations, Proc. 4th ann. Princeton Conf. on Inf. Sciences and Systems (1976) 169-176.

[5] Scott, D.: Datatypes as Lattices, Lecture Notes in Math 499 (1976). also SIAM J Comput 5 (1976) 3, 522-587.

[6] Reynolds, J.C.: Definitional Interpreters for higher order programming languages, Proc. ACM 27th Math. Conf. (1972) 717-740.

[7] Hoare, C.A.R.and Wirth, N.: An Axiomatic Definition of the Programming Language PASCAL, Acta Inf. 2 (1973), 335-355.

[8] Tennent, R.D.: The Denotational Semantics of Programming Languages, CACM 19 (1976) 8, 437-453.

[9] Mosses, P.: The Mathematical Semantics of Algol 60, Tech. Mon. Oxford Univ., Comp. Lab., PRG-12, (Jan. 1974).

[10] Ligler, G.T.: Proof Rules, Mathematical Semantics and Language Design, D.Ph. Thesis, Oxford Univ. (1975).