AFDELING INFORMATICA   IW 22/74   SEPTEMBER

J.W. DE BAKKER

LEAST FIXED POINTS REVISITED

Prepublication

Least fixed points revisited [*]

by

J.W. de Bakker

ABSTRACT

Parameter mechanisms for recursive procedures are investigated. Contrary to the view of Manna et al., it is argued that both call-by-value and call-by-name mechanisms yield the least fixed points of the functionals determined by the bodies of the procedures concerned. These functionals differ, however, according to the mechanism chosen. A careful and detailed presentation of this result is given, along the lines of a simple typed lambda calculus, with interpretation rules modelling program execution in such a way that call-by-value determines a change in the environment and call-by-name a textual substitution in the procedure body.

---

[*] This paper is not for review; it is meant for publication elsewhere.

CONTENTS

NOTATION

Section 2

$s, t, t_i, t', \ldots$	individual	
$\sigma, \tau, \tau_j, \tau', \ldots$	function	terms
$S, T, \ldots$	functional	

$p, p', \ldots$		individual	
$\pi, \ldots$	boolean	function	terms
$P, \ldots$		functional	

$a \in A, \; \alpha \in A$
$b \in B, \; \beta \in B$ constants

$x, y, z, u \in X, \; \xi, \eta \in X$
$q \in Q, \; \chi \in Q$ variables

$\phi \in F, \; \psi \in G$ procedure symbols

$\tau(t_1, \ldots, t_n), \; \pi(t_1, \ldots, t_n)$
$T(\tau_1, \ldots, \tau_r), \; P(\pi_1, \ldots, \pi_r)$ application

$\nu x_1 \ldots x_\ell \lambda y_1 \ldots y_m \cdot t,$
$\nu x_1 \ldots x_\ell \lambda y_1 \ldots y_m \cdot p,$ abstraction
$\lambda \xi_1 \ldots \xi_r \cdot \tau, \; \lambda \chi_1 \ldots \chi_r \cdot \pi$

$\underline{if} \; p \; \underline{then} \; t' \; \underline{else} \; t''$
$\underline{if} \; p \; \underline{then} \; p' \; \underline{else} \; p''$ selection

$n(\tau), \; \langle n(T), r(T) \rangle$ rank

$1 \leq i \leq n, \; 1 \leq j \leq r,$
$1 \leq h \leq \ell, \; 1 \leq k \leq m$ indices

$\bar{t}, \bar{\tau}, \bar{x}, \bar{\xi}, \ldots$ vector notation

$\equiv$ syntactic identity between terms

2

Section 3

$s[t/x]$, $s[\tau/\xi]$,... substitution

$E_{vf}^{x\xi}$, $E_{v}^{\bar{x}}$,... change of environment

Section 4

$V = V_0 \cup \{\perp\}$ domain

C constant-interpretation

E variable-interpretation (environment)

$\mathcal{D}$ declarations

$J = \langle V, C, E; \mathcal{D} \rangle$ interpretation

$v, w \in V$, $f \in V^n \to V$,
$F \in (V^n \to V)^r \to (V^n \to V)$ } elements of (higher) domains

$valt(t, J, N)$
$val(t, J)$, $val(t, E)$ } evaluation functions

Section 5

$v_1 \subseteq v_2$, $f_1 \subseteq f_2$, $F_1 \subseteq F_2$ partial orderings

$val(\tau, J)$, $val(T, J)$ extension of val

$t_1 \subseteq t_2$, $\tau_1 \subseteq \tau_2$, $T_1 \subseteq T_2$ atomic formulae

Φ, Ψ sets of atomic formulae

ρ element of Φ

$\Phi \models_{\mathcal{D}} \Psi$ assertion

$\Phi[V, C, E; \mathcal{D}]$ $\langle V, C, E; \mathcal{D} \rangle$ satisfies Φ

$\Phi \models \Psi$, $\models \Psi$, $\models t_1 = t_2$,... simplified forms of assertions

Section 7

ϕ_0 the nowhere defined procedure

ω $\phi_0(\bar{z})$

Ω $\nu\bar{x}\lambda\bar{y}.\omega$

0 $\lambda\bar{\xi}.\Omega$

$\tilde{t},\tilde{\tau},\tilde{T}$ replacing procedure symbols by new variables

$t^{(i)},\tau^{(i)},T^{(i)}$ approximants to t,τ,T

1. INTRODUCTION

1.1. MOTIVATION

The fixed point approach in the semantics of programming languages has gained considerable popularity in recent years. The basic programming notions of recursion and iteration have found a satisfactory *mathematical* treatment in terms of least fixed points, as opposed to the *operational* methods, where the emphasis is on techniques using stacks, displays and the like. (For a discussion on the distinction between mathematical and operational semantics one should consult the works of SCOTT and STRACHEY, such as [20].) In order to explain our motivation for the present paper, a brief sketch of the history of the subject is needed.

KLEENE's first recursion theorem [6] already gave a characterization of recursive functions (albeit restricted to integer functions with parameters called-by-value, see below) in terms of least fixed points. For some time, applications in programming theory remained tentative, however. We refer for example to the results of McCARTHY [10], and, in particular, to the early work of LANDIN (e.g. [7]) where CURRY's Y combinator was used to deal with recursion in such a way that the fact that the fixed points concerned are *least* with respect to a suitable partial ordering remained implicit. As another important predecessor we mention MORRIS [12]. In 1969, a number of people arrived independently at some methods and results causing a revival of the fixed point approach, viz. BEKIC, PARK [15] and SCOTT and DE BAKKER [19]. To be more specific, by "fixed point approach" we refer to the whole of techniques for proving properties of programs which take as starting point the fact that the function defined by a recursive procedure can, in a sense to be made precise presently, be viewed as the least fixed point of a functional which is associated in a rather natural way with the body of the procedure declaration. The paper [19] also contained the first statement of an important rule of proof, SCOTT's induction rule, which has found a variety of applications in the next few years [1,2,3,4,8,9,11]. On the theoretical side, the invention of SCOTT's models of the lambda calculus, where the relationship between the least fixed point result and the Y combinator could be settled, has added to the success of the method.

As to the applications, we are in particular interested in the papers by MANNA and his colleagues - then at Stanford - which contain both a long list of examples, and a discussion of the relationship between SCOTT's induction (computation induction, as they call it), and other methods such as MORRIS' truncation induction [13].

However, these papers contain also the first statement of a view which, in our opinion, has been the cause (and effect) of some confusion, viz. that "many programming languages use implementations (such as call-by-value) which do not necessarily lead to the least fixed point" (MANNA [9], p.529), or (MANNA in [8], p.496) "... we are interested in computation rules that yield the least fixed point ... we call such computation rules fixed point computation rules ... the left-most innermost rule is not a fixed point rule...". In both of these papers, the work of MORRIS [12] is mentioned in support of the quoted statements. However, the main assertion of MORRIS seems to us to be that the function determined by a procedure with parameters called-by-value may be properly included in the function determined by the same procedure with parameters called-by-name. (A partial function f is said to be included in a partial function g (f $\subseteq$ g) iff whenever f is defined, g is defined with the same value.) It is *not* asserted by MORRIS that procedures with parameters called-by-value do not yield the least fixed point (and it would have been rather surprising when he had done so, since it is the very content of KLEENE's first recursion theorem, which MORRIS quotes at length, that recursion with call-by-value *does* lead to the least fixed point!) The least fixed point result for call-by-name parameters is left open by MORRIS, i.e. only a conjecture to that effect is formulated.

In order to explain what we see as the cause of the confusion in MANNA [8,9], consider the example of a recursive procedure (from MANNA [9]):

(1.1) $\phi(x,y) \Leftarrow \underline{if}\ x = 0\ \underline{then}\ 0\ \underline{else}\ \phi(x-1,\phi(x,y))$

Suppose we consider ϕ for integer x,y. Then, if ϕ has parameters called-by-value, we obtain for the function f determined by (1.1), say f_v:

$f_v(x,y) = \underline{if}\ x = 0\ \underline{then}\ 0\ \underline{else}\ undefined$

whereas, when the parameters are called-by-name, we obtain f_n:

$$f_n(x,y) = \underline{if}\ x \geq 0\ \underline{then}\ 0\ \underline{else}\ undefined.$$

Clearly, $f_v \subseteq_{\not\leftarrow} f_n$. Writing

(1.2) $\phi(x,y) \Leftarrow \Phi(\phi)(x,y)$

as short-hand for (1.1), with Φ determining a functional F, it is now certainly impossible that f_v and f_n are both *the* least fixed point of F. *However, the notation in (1.2) leaves out the important distinction between the two parameter mechanisms used,* since one same Φ is used for both cases. What is needed is a treatment of (1.2) such that the functional term Φ carries the information about the parameters along: We then have *two* declarations: $\phi(x,y) \Leftarrow \Phi_v(\phi)(x,y)$, and $\phi(x,y) \Leftarrow \Phi_n(\phi)(x,y)$, which determine functions f_v and f_n, and functionals F_v and F_n, such that f_v is the least fixed point of F_v, and f_n is the least fixed point of F_n (and that $f_v \subseteq f_n$, with the possibility that $f_v \subseteq_{\not\leftarrow} f_n$).

Now we can state the goal of our paper: We want to make the above considerations precise, and to prove, in careful detail, the least fixed point result for both parameter mechanisms. Of course, we do not claim any essential originality for this result. E.g., the same result is obtained by RO-SEN [17], and it is also contained in the lecture notes of NIVAT [14]. An attempt at clarification of the same issues was made by DE ROEVER [16], who also emphasizes that different parameter mechanisms give rise to different transformations, but elaborates this idea in a framework of axiomatized relations. An elementary exposition, which does not involve the somewhat advanced logical and algebraic tools of ROSEN [17] and NIVAT [14], but which has certainly benefited from the ideas of the lambda calculus, may be of interest. No applications are dealt with; we do not even prove the continuity of the functionals, nor do we give a justification of SCOTT's induction rule. On the contrary, we concentrate solely on the stated problem. We shall use a rather extensive formalism for this purpose, and spend much attention to a detailed development of the argument. We feel that this may be justified in a situation where we have set out to clarify a confusion which, some-

what surprisingly, seems rather widespread.

1.2. OUTLINE OF THE PAPER

In section 2 we define the syntax of a formal language centered around the notions of *application, abstraction, recursion* and *selection*. In spirit, the syntax is very much like a typed lambda calculus, with two major differences

- the explicit addition of recursion by procedure declarations and - calls
  (as opposed to the implicit recursion via the Y-combinator in the non-
  typed lambda calculus)

- the explicit notational distinction between call-by-value and call-by-name
  parameters

and one minor difference: the number of types is restricted to three: individual, function, and functional.

In section 3 we give the standard definitions of free and bound (occurrences of) variables, and of substitution.

Section 4 is of central importance, giving the semantics of the constructs in our language. Terms are provided with an interpretation: select a domain V, map constants and variables to elements of V (and of the derived domains of higher type), and, moreover, fix a set of declarations for the procedure symbols. This being done, a process of evaluation is prescribed: The well-known extension of V with an extra element to provide a "value" in the case of non-terminating evaluations is used, and, for terminating evaluations, the number of steps needed is carried along (this being of importance in a later proof which uses induction on this number). Recursion is defined by body replacement, call-by-value parameters by changing the environment, and call-by-name parameters by substitution.

Section 5 leads up to the formalism to state facts about our terms which hold for all interpretations (so-called valid assertions). First a partial order on the domains is introduced, and various properties of the val-function are derived, which are of technical importance for the proof of the monotonicity theorem in section 6.

Section 7 introduces the notion of approximating a term by procedure-free approximants, and develops a precise notation for this, resulting in

the lemma that, for each term t, its value in a given interpretation is also the value of an approximant to t.

Section 8, finally, brings the proof of the least fixed point theorem, which relies heavily on the result of section 7.

ACKNOWLEDGMENT. *Our interest in applying a formalism near to the lambda calculus in investigating semantics has been stimulated by lecture notes of Robin Milner, explaining work of Peter Landin and Dana Scott.*

2. SYNTAX

We introduce a formal language which contains the main programming concepts relevant for our purpose. Starting from some initial classes of expressions, construction rules are provided to build up more complex expressions. These rules correspond to the following programming concepts:

- *application*   apply a function to one or more arguments
- *abstraction*   an expression may be "parametrized", yielding a function
                    of one or more arguments (such abstraction is part of the
                    mechanism usually invoked at procedure declaration)
- *recursion*   a mechanism for "declaring" and "calling" (possibly re-
                    cursive) procedures is introduced
- *selection*   this gives the usual conditional construct in programming

A subset of the expressions of our language, with elements called *terms*, is intended as the class of "abstract programs". In section 4, a method will be given to *interpret* these terms by means of an evaluation mechanism yielding *individuals, functions* and *functionals* as values. The evaluation mechanism is, of course, designed in such a way that it models program execution, in sofar as this is concerned with the programming concepts just mentioned. Anticipating the precise definitions, we already indicate that an interpretation will start with the choice of a domain V, such that the terms of our language will, in this interpretation, have values according to the following table:

Class of terms	Denotation	Intended interpretation
individual	$s,t,t_0,t_i,t',\ldots$	$\in V$
function	$\sigma,\tau,\tau_0,\tau_j,\tau',\ldots$	$\in V^n \to V$
functional	$S,T,T',\ldots$	$\in (V^n \to V)^r \to (V^n \to V)$
boolean	$p,p',\ldots$	$\in \{0,1\}$
boolean function	$\pi,\ldots$	$\in V^n \to \{0,1\}$
boolean functional	$P,\ldots$	$\in (V^n \to \{0,1\})^r \to (V^n \to \{0,1\})$

Figure 1. Intended interpretation of terms

The terms are made up by means of the construction rules mentioned above starting from certain given symbol classes, of *constants, variables,* and *procedure symbols.*

Type of class		Notation	
		Class	Element
Individual $\Big\rangle$		A	a
Function $\Big\rangle$ constants		A	α .
Boolean		B	b
Boolean function		B	β
Individual		X	x,y,z,u
Function $\Big\rangle$ variables		X	ξ,η
Boolean		Q	q
Boolean function		Q	χ
Procedure $\Big\rangle$ symbols		F	ϕ
Boolean procedure		G	ψ

Figure 2. The initial classes of symbols

The syntax of our language is now given in the following two tables (explanatory remarks follows the definition):

term \ operation	*constant*	*variable*	*application*
t (individual)	$a \in A$	$x \in X$	$\tau(t_1,\ldots,t_n)$
p	$b \in B$	$q \in Q$	$\pi(t_1,\ldots,t_n)$
τ (function)	$\alpha \in A$	$\xi \in X$	$T(\tau_1,\ldots,\tau_r)$
π	$\beta \in B$	$\chi \in Q$	$P(\pi_1,\ldots,\pi_r)$
T (functional)	—	—	—
P	—	—	—

Figure 3a. Syntax, first part

term \ operation	*abstraction*	*selection*	*recursion*
t (individual)	—	<u>if</u> p <u>then</u> t' <u>else</u> t"	—
p	—	<u>if</u> p <u>then</u> p' <u>else</u> p"	—
τ (function)	$\nu x_1 \ldots x_\ell \lambda y_1 \ldots y_m \cdot t$ ($\ell+m>0$)	—	$\phi \in F$
π	$\nu x_1 \ldots x_\ell \lambda y_1 \ldots y_m \cdot p$	—	$\psi \in G$
T (functional)	$\lambda \xi_1 \ldots \xi_r \cdot \tau$ ($r>0$)	—	
P	$\lambda \chi_1 \ldots \chi_r \cdot \pi$	—	

Figure 3b. Syntax, second part

We assume – without bothering to justify this – that each term can be uniquely parsed. Moreover, parentheses will be used freely to enhance readability.

Syntactic identity between terms will be denoted by "$\equiv$". Some examples of terms are:

1. Individual terms:

    a, x, $\phi(x_1,x_2)$, $(\lambda x.x)(a)$, <u>if</u> p <u>then</u> a <u>else</u> $\phi(\alpha(x_1)$, $\phi(x_1,x_2))$

2. Function terms:

    α, ξ, $(\lambda\xi.\alpha)(\phi)$, $\nu x_1 x_2$. <u>if</u> p <u>then</u> a <u>else</u> $\phi(\alpha(x_1)$, $\phi(x_1,x_2))$

    $\lambda y_1 y_2$. <u>if</u> p <u>then</u> a <u>else</u> $\phi(\alpha(y_1)$, $\phi(y_1,y_2))$

    (we adopt here the obvious conventions for $\ell = 0$ or m = 0)

3. Functional terms:

    $\lambda\xi$. $\nu x_1 x_2$. <u>if</u> p <u>then</u> a <u>else</u> $\xi(\alpha(x_1)$, $\xi(x_1,x_2))$

    $\lambda\xi$. $\pi y_1 y_2$. if p. then a else $\xi(\alpha(y_1)$, $\xi(y_1,y_2))$

    (for suitable interpretation of p,a and α, these two functional terms
    correspond to the Φ_v and Φ_n of the introduction, cf. remark 2 below).

The following remarks will help the reader in reading and understand-
ing the syntactic definitions in figures 3a, 3b.

1. (Reading the tables). Consider, e.g., the first line (after t in fig. 3a).
   This should be read as:
   - Each individual constant or individual variable is an individual term.
   - If τ is a function term, and each t_i, i=1,...,n, is an individual term,
     then $\tau(t_1,...,t_n)$ is an individual term (obtained by the construction
     rule of application).

2. (Abstraction). Most of the tables should now be readable, apart from the
   abstraction-column, which needs further explanation. Consider the con-
   struct $\tau \equiv \nu x_1...x_\ell \lambda y_1...y_m.t$. For each individual term t, τ denotes a
   function term which, in the interpretation to be given presently, will
   obtain as meaning a function with
   - the $\ell \geq 0$ arguments (formal parameters) $x_1,...,x_\ell$ *called-by-value*
   - the m ≥ 0 arguments (formal parameters) $y_1,...,y_m$ *called-by-name*.
   In other words, the ν-abstraction is intended to model call-by-value
   parametrization, the λ-abstraction call-by-name parametrization. (The
   reader should not confuse this statement of intention, to be made pre-
   cise in section 4, with the "normal" λ-abstraction in the lambda calcu-
   lus, where the conversion order is (very much) left open.) Again antici-
   pating, call-by-value parameters will be evaluated by changing the en-
   vironment, i.e. the variable-value correspondence, and call-by-name
   parameters by textual substitution. For these definitions to make sense,

we assume from now on that the x_n, $1 \le h \le \ell$, y_k, $1 \le k \le m$, are all different variables. An analogous requirement is imposed upon the ξ_j, $j=1,\ldots,r$, in the formation of $\lambda \xi_1 \ldots \xi_r.\tau$.

3. (Functionals). For functional terms, we do not need the call-by-value type of arguments, and we restrict ourselves to the usual λ-abstraction, thus turning function terms into functional terms. It will be noticed that only very limited means are provided to construct functional terms. As a matter of fact, they are, strictly speaking, unnecessary, and are only introduced to obtain eventually a more appealing form of the least fixed point theorem. Note also that we introduce functionals of a rather restricted format: Instead of elements of $[(V^{n_1} \to V) \times (V^{n_2} \to V) \times \ldots \times (V^{n_r} \to V)] \to$ $\to (V^n \to V)$, we have the simple form as given. This is a restriction imposed for convenience sake only. All the results of the paper go through for the more general case, but we did not want to add an extra burden to the already rather heavy formalism.

4. (Recursion). The recursion column is as yet rather meagre: no declarations are given yet, only the procedure symbols. We find it more convenient to introduce declarations as part of the interpreting mechanism, though an appraoch which brings in declarations at an earlier stage might also have been adopted.

5. (Rank and arity). The syntax tables are not very explicit on the role of the integers n and r. The following supplementary information is in order: Each function term τ has a certain so-called rank $n(\tau)$, each functional term T has a rank-pair $<n(T),r(T)>$. The rank is initially given for the α, ξ and ϕ, and for the other constructs it is defined as follows
   - if $\tau \equiv \nu x_1 \ldots x_\ell \lambda y_1 \ldots y_m.t$, then $n(\tau) = \ell + m$
   - if $\tau \equiv T(\tau_1,\ldots,\tau_r)$, and $n(\tau_i) = n (i=1,\ldots,r)$, then $n(\tau) = n$
   - if $T \equiv \lambda \xi_1 \ldots \xi_r.\tau$, and $n(\tau) = n$, then $n(T) = n$ and $r(T) = r$
   - similar definitions hold for the boolean case.
   Furthermore, we require
   - if $t \equiv \tau(t_1,\ldots,t_n)$, then $n(\tau) = n$
   - if $\tau \equiv T(\tau_1,\ldots,\tau_r)$, then $n(T) = n(\tau_i)$, $i=1,\ldots,r$, and $r(T) = r$.
   This system is, of course, designed in this way in order that
   - each τ of rank $n = n(\tau)$ is to be interpreted as a function of arity

n: $V^n \to V$

- each T of rank $<n,r>$ is to be interpreted as a functional:

$$(V^n \to V)^r \to (V^n \to V).$$

- similarly for the boolean case.

We are aware of the fact that we have not adopted here the most general solution. We have envisaged a system with terms t of rank $n(t) \geq 0$, with abstraction restricted to one variable, such that, for $t \equiv \lambda x.t'$ or $t \equiv \nu x.t'$, $n(t) = n(t')+1$, and $n(t) = 0$ indicating that t is to be interpreted as an individual ($\in V$), and with application restricted to the one-argument case. For our present purpose, the syntax as in figs. 3a,b was thought to be preferable. The restriction of our presentation to terms of three levels (individual, function, functional) is closer to the concepts as they appear in programming than a system with an infinite hierarchy, and our main goal - clarification of the two parameter mechanisms - seems, after some experiments with the mathematically more elegant approach just sketched, to be achieved in a better way.

One last remark on notation: We sometimes use a "vector"-notation, and write $\tau(\bar{t})$ for $\tau(t_1,\ldots,t_n)$, $T(\bar{\tau})$ for $T(\tau_1,\ldots,\tau_r)$, $\bar{x}$ for $(x_1,\ldots,x_\ell)$, etc.

3. SUBSTITUTION

In the interpretation of terms to be given in section 4, we shall define the evaluation of call-by-name parameters by a process of textual replacement of "formal" by "actual" parameters, i.e., by means of *substitution*. Therefore, we devote this section to a precise definition of this operation. We do this by a restatement of standard techniques, see e.g. [5], as adopted to our present goals. At the end of the section a notation for changing the environment - which is used to model call-by-value evaluation - is given.

First we introduce the notion of a *variable occurring in a term* in

<u>DEFINITION</u> 3.1 (Occurrences).

1. x occurs in t iff

    1.1. t ≡ x

    1.2. t ≡ $\tau(t_1,\ldots,t_n)$, and x occurs in τ or any of the t_i, i=1,...,n.

    1.3. t ≡ <u>if</u> p <u>then</u> t' <u>else</u> t", and x occurs in p, t' or t".

2. x occurs in τ iff

    2.1. τ ≡ $\nu\bar{x}\lambda\bar{y}.t$, and x occurs in t

    2.2. τ ≡ $T(\tau_1,\ldots,\tau_r)$, and x occurs in T or any of the τ_j, j=1,...,r.

3. x occurs in T iff

    3.1. T ≡ $\lambda\bar{\xi}.\tau$, and x occurs in τ.

4. ξ occurs in t iff

    4.1. t ≡ $\tau(t_1,\ldots,t_n)$, and ξ occurs in τ or any of the t_i, i=1,...,n.

    4.2. t ≡ <u>if</u> p <u>then</u> t' <u>else</u> t", and ξ occurs in p, t' or t".

5. ξ occurs in τ iff

    5.1. τ ≡ ξ

    5.2. τ ≡ $\nu\bar{x}\lambda\bar{y}.t$, and ξ occurs in t

    5.3. τ ≡ $T(\tau_1,\ldots,\tau_r)$, and ξ occurs in T or any of the τ_j, j=1,...,r.

6. ξ occurs in T iff

    6.1. T ≡ $\lambda\bar{\xi}.\tau$, and ξ occurs in τ.

7. The definitions for p, π or P are similar.

    Observe that x does not occur in λx.a. Next, we need the notions of *bound* and *free* occurrences of a variable in a term.

<u>DEFINITION</u> 3.2 (Bound and free occurrences).

1. An occurrence of a variable x in a term is bound iff x occurs in a part of that term of the form $\nu x_1\ldots x_\ell\lambda y_1\ldots y_m.t$, with x ≡ x_h, for some h, $1 \le h \le \ell$, or x ≡ y_k, for some k, $1 \le k \le m$.

2. An occurrence of a variable x in a term is free, otherwise.

3. An occurrence of a variable ξ in a term is bound iff ξ occurs in a part of that term of the form $\lambda\xi_1,\ldots,\xi_r.\tau$, with $\xi \equiv \xi_j$, for some j, $1 \le j \le r$.

4. An occurrence of a variable ξ in a term is free, otherwise.

EXAMPLES:

- x occurs bound in λx.x, free in α(x), and bound *and* free in (λx.x)(α(x)).
- both ξ and x occur bound in λξ.λx.α₁((λη.ξ)(η)(x)), whereas η occurs free in that same term.

We now define the important notion of substitution. A term t (or τ) may be substituted for (i.e. replace all free occurrences of) a variable x (or ξ) in any term s,σ,S,p,π or P. The results are denoted by s[t/x],...
...,P[t/x], s[τ/ξ],...,P[τ/ξ]. The process of substitution is defined by induction on the complexity of the terms involved:

DEFINITION 3.3 (Substitution).

1. s[t/x]
   1.1. a[t/x] ≡ a, x[t/x] ≡ t, y[t/x] ≡ y (y≠x).
   1.2. σ(s̄)[t/x] ≡ σ[t/x](s̄[t/x]).
   1.3. (if p then s' else s")[t/x] ≡ if p[t/x] then s'[t/x] else s"[t/x].

2. σ[t/x]
   2.1. α[t/x] ≡ α, ξ[t/x] ≡ ξ, φ[t/x] ≡ φ.
   2.2. (νx̄λȳ.s)[t/x]
       ≡ νx̄λȳ.s, if x ≡ x_h, for some h, $1 \le h \le \ell$, or x ≡ y_k, for some k, $1 \le k \le m$; otherwise
       ≡ νx̄λȳ.s[t/x], if none of the x̄ or ȳ occurs free in t; otherwise
       ≡ νz̄λū.s[z̄/x̄][ū/ȳ][t/x] where the z̄,ū are *new* variables
   2.3. S(σ̄)[t/x] ≡ S[t/x](σ̄[t/x])

3. S[t/x]
   3.1. (λξ̄.τ)[t/x] ≡ λξ̄.τ[t/x]

4. s[τ/ξ], σ[τ/ξ], S[τ/ξ]. We only give the central cases, the remaining ones then follow as in 1-3.
   4.1. ξ[τ/ξ] ≡ τ, η[τ/ξ] ≡ η (η≠ξ)
   4.2. (λξ̄.σ)[τ/ξ]
       ≡ λξ̄.σ, if ξ = $ξ_j$, for some j, $1 \le j \le r$; otherwise
       ≡ λξ̄.σ[τ/ξ], if none of the $ξ_j$ occurs free in τ; otherwise
       ≡ λη̄.σ[η̄/ξ̄][τ/ξ] where the η̄ are *new* variables

5. Substitution in boolean terms is defined similarly.

REMARK: The precautions in 2.2 and 4.2 have the usual reason: Without them, variables free in t (or τ) would be turned into bound variables (*then*, e.g., $(\lambda y.x)[y/x] \equiv \lambda y.y)$, and substitution would not be "meaning preserving" (e.g., in the intended interpretation, $\lambda y.x$ determines a function which, for each argument, yields the value of x as a result, whereas $\lambda y.y$ determines the identity function).

EXAMPLES OF SUBSITUTION:

1. $(\lambda\xi.\lambda x.\alpha_1((\lambda\eta.\xi)(\eta)(x)))[\lambda z.\xi(a)/\eta] \equiv \lambda\zeta.\lambda x.\alpha_1((\lambda\eta.\zeta)((\lambda z.\xi(a))(x)))$

2. $((\lambda y.x)(\phi(x)))[\xi(y)/x] \equiv (\lambda z.\xi(y))(\phi(\xi(y)))$

The following lemma states a number of basic properties of substitutions to be used in later sections:

LEMMA 3.4.

1. *If* y *is not free in* s, *and neither* y *nor any variable free in* t *is bound in* s, *then* s[y/x][t/y] $\equiv$ s[t/x].

2. *If* y $\neq$ x, *and* y *is not free in* t", *and no variable free in* t' *or* t" *is bound in* s, *then* s[t'/y][t"/x] $\equiv$ s[t"/x][t'[t"/x]/y].

PROOF. See HINDLEY, LERCHER & SELDIN [5]. □

Call-by-name parameters are dealt with by means of substitution, call-by-value parameters by changing the environment. A definition of this follows in the next section, but we already introduce a notation designed for this purpose.

Let, for the moment, E be any function mapping arguments x to values v, and arguments ξ to values f. Then $E^{x\xi}_{vf}$ is a function which satisfies:

1. $(E^{x\xi}_{vf})(x) = v$, $(E^{x\xi}_{vf})(\xi) = f$.

2. $(E^{x\xi}_{vf})(y) = E(y)$, for each y $\neq$ x.

   $(E^{x\xi}_{vf})(\eta) = E(\eta)$, for each $\eta \neq \xi$.

Extension of the notation to the vector case: $E^{\overline{x\xi}}_{\overline{vf}}$, and restriction of it to cases such as E^{x}_{v}, should be clear. Observe that a notation such as $E^{x\xi}_{vf}$ has nothing to do with substitution which is a notion making sense only for *linguistic* entities.

4. SEMANTICS

The variety of terms as introduced in section 2 are now provided with a meaning. We define a process of interpretation of terms, in which the notion of their evaluation - by means of the function val - plays a central part.

An interpretation $J = <V,C,E;\mathcal{D}>$ has the following components:

1. A domain (non-empty set) V.

2. C (dealing with the *constants*) maps A to V, B to $\{0,1\}$, A to $V^n \to V$, B to $V^n \to \{0,1\}$ (with, for $\alpha \in A$, $n = n(\alpha)$, etc.).

3. E (dealing with the *variables*) maps X to V, Q to $\{0,1\}$, X to $V^n \to V$ and Q to $V^n \to \{0,1\}$ (with, for $\xi \in X$, $n = n(\xi)$, etc.) (The variable-value mapping established by E is often referred to as the *environment*.)

4. $\mathcal{D}$ (dealing with the *declarations*) maps procedure symbols ϕ (or boolean procedure symbols ψ) of rank n to function terms τ (or boolean function terms π) of the same rank.

We now discuss the way in which the interpretations J are used to obtain values of terms: A certain computational process is defined, which is intended to model the semantics of the programming concepts concerned - such as described e.g. in the ALGOL 60 report - and which, for each of the pairs $<t,J>$ and $<p,J>$ will yield a value in V or $\{0,1\}$, respectively, as their *value*. However, this gives rise to an important point: We know that some computations in a programming language with recursion do not terminate, and, hence, that our function val will have to be *partial*: for some terms, no value will be delivered. In order to deal with this problem, the domain V is *extended* with one special element $\perp$, which is not an element of V, and which stands for "undefined". From now on, V will refer to this extended set, and the subset of all "defined" elements of V will be called V_0; i.e., we have $V = V_0 \cup \{\perp\}$. This extension is seemingly a trick which does not do away with any of the essential problems stemming from possibly unending computations. However, it will turn out to lead to a streamlining of much of the ensuing argument, and may be compared to some extent with the introduction of ∞ in the calculus.

We shall next define the function val, with $val(t,J)$ yielding $v \in V$,

according to the following scheme: First we introduce the *partial* function $valt(t,J,N)$ (*valt* standing for terminating evaluation), where N is an integer which tells us how many computation steps are needed in order to arrive at the result v. Then we define the *total* function $val(t,J)$ in terms of $valt(t,J,N)$.

DEFINITION 4.1 (Terminating evaluations).

Let $J = <V,C,E;D>$ be an interpretation, and t an individual term. $valt(t,J,N)$ is defined by the following inductive definition:

1. $t \equiv a \in A$.

   If $C(a) = v \in V_0$, then $valt(a,J,1) = v$.

2. $t \equiv x \in X$.

   If $E(x) = v \in V_0$, then $valt(x,J,1) = v$.

3. $t \equiv \tau(t_1,\ldots,t_n) \equiv \tau(\bar{t})$.

   3.1. $\tau \equiv \alpha \in A$

   If $valt(t_i,J,N_i) = v_i \in V_0$, $i=1,\ldots,n$, and $C(\alpha)(\bar{v}) = v \in V_0$, then

   $$valt(\alpha(\bar{t}),J,(\textstyle\sum_i N_i)+1) = v.$$

   3.2. $\tau \equiv \xi \in X$

   If $valt(t_i,J,N_i) = v_i \in V_0$, $i=1,\ldots,n$, and $E(\xi)(\bar{v}) = v \in V_0$, then

   $$valt(\xi(\bar{t}),J,(\textstyle\sum_i N_i)+1) = v.$$

   3.3. (The central case). $\tau \equiv \nu\bar{x}\lambda\bar{y}.t$

   If $valt(t_h,J,N_h) = v_h \in V_0$, $h=1,\ldots,\ell$,

   and

   $$valt(t[z_h/x_h]^\ell_{h=1}[t_{\ell+k}/y_k]^m_{k=1}, <V,C,E^{\bar{z}}_{\bar{v}};D>,N) = v$$

   where the $\bar{z} = (z_1,\ldots,z_\ell)$ are *new* variables,

   then

   $$valt((\nu\bar{x}\lambda\bar{y}.t)(\bar{t}),J,(\textstyle\sum_h N_h)+N) = v.$$

   3.4. $\tau \equiv T(\tau_1,\ldots,\tau_r) = T(\bar{\tau})$, where $T \equiv \lambda\bar{\xi}.\tau_0$.

   If $valt(\tau_0[\bar{\tau}/\bar{\xi}](\bar{t}),J,N) = v$

   then

   $$valt(T(\bar{\tau})(\bar{t}),J,N+1) = v.$$

3.5. $\tau \equiv \phi \in F$

    If $valt(\mathcal{D}(\phi)(\bar{t}),J,N) = v$

    then

        $valt(\phi(\bar{t}),J,N+1) = v.$

4. $\tau \equiv \underline{if}\ p\ \underline{then}\ t'\ \underline{else}\ t''.$

    If $valt(p,J,N) = 1$ and $valt(t',J,N') = v$, then

        $valt(\underline{if}\ p\ \underline{then}\ t'\ \underline{else}\ t'',J,N+N') = v.$

    If $valt(p,J,N) = 0$ and $valt(t'',J,N'') = v$, then

        $valt(\underline{if}\ p\ \underline{then}\ t'\ \underline{else}\ t'',J,N+N'') = v.$

5. The definition of $valt(p,J,N)$ is completely analogous to 1-4 and omitted.

DEFINITION 4.2 (Evaluations).

1. $val(t,J) = v$ if there exists N such that $valt(t,J,N) = v.$

2. $val(t,J) = \bot$, otherwise.

3. Similarly for $val(p,J)$.

    The following remarks have to be made on these definitions:

1. Observe that, if $valt(t,J,N) = v$, then $v \in V_0$. This follows by induction on the complexity of t. All the "terminal" cases in the inductive definition explicitly require that v be an element of V_0, and this property is inherited by the "non-terminal" cases.

2. (Clauses 1,2 of def. 4.1). The case that t is a constant or variable are clear.

3. (Clauses 3.1, 3.2 of def. 4.1). Let $t = \tau(\bar{t})$ be an application, with τ a constant or variable. Here we require that, for $valt$ to be defined, v and each of the v_i, i=1,...,n, be in V_0. Observe that, otherwise, $val(t,J)$ will, by definition 4.2, be set to $\bot$. The requirement that v_i be in V_0 is justified by our desire to have that our *basic* functions (i.e., the functions that are not defined via our language) satisfy the property that a function value be undefined when one of its arguments is undefined. That $v \in V_0$ fits in with our scheme that $valt$ defines only terminating computations.

4. (The central case). Let $t \equiv \tau(\bar{t})$, with τ an abstraction. Here we observe that

    a. The value parameters t_h, h=1,...,ℓ, are evaluated first. The fact that

the v_h are an outcome of *valt* guarantees that these evaluations terminate (cf. remark 1). Note that, if *valt* would not terminate for some t_h, then the attempt at defining *valt* for t would fail, and clause 2 of definition 4.2 would apply.

b. The environment is changed to a new environment which links new variables z_h to the v_h obtained above. The need for this change of variables from $\bar{x}$ to $\bar{z}$ is explained by the possibility that the $\bar{x}$ occur free in the $t_{\ell+k}$.

c. For the name parameters, no evaluation takes place, but a process of substituting the actual parameters $t_{\ell+k}$ for the formal parameters y_k, k=1,...,m, is instead applied.

5. (Clause 3.4 of def. 4.1). This case is dealt with only for completeness sake.

6. (Clause 3.5 of def. 4.1). Here we find the rule of body replacement, which gives the standard meaning to recursion: The procedure symbol ϕ is replaced by the term $\mathcal{D}(\phi)$ which forms its body, and, next, the evaluation is continued.

7. (Clause 4 of def. 4.1). This defines the standard meaning of conditionals.

5. ASSERTIONS

Before we introduce the formalism to assert that certain facts hold for our terms under all interpretations, we need some preparatory concepts and lemma's.

Firstly, we introduce a partial ordering on our domains:

DEFINITION 5.1.

1. For $v_1, v_2 \in V$,
   $v_1 \sqsubseteq v_2$ iff $v_1 = \bot$ or $v_1 = v_2$.

2. For $f_1, f_2 \in V^n \to V$,
   $f_1 \sqsubseteq f_2$ iff, for all $\bar{v} \in V^n$, $f_1(\bar{v}) \sqsubseteq f_2(\bar{v})$.

3. For $F_1, F_2 \in (V^n \to V)^r \to (V^n \to V)$,
   $F_1 \sqsubseteq F_2$ iff, for all $\bar{f} \in (V^n \to V)^r$, $F_1(\bar{f}) \sqsubseteq F_2(\bar{f})$.

Clearly, "$\subseteq$" is indeed a partial ordering. Thus $v_1 = v_2$ iff $v_1 \subseteq v_2$ and $v_2 \subseteq v_1$, etc. Anticipating again, now that $\subseteq$ is defined on V, we know what $val(t,J) \subseteq val(t',J)$ means, viz. that, for this J, either the evaluation of t does not terminate, or, t and t' have the same value in V_0. If the inclusion and its reverse hold for all J, we shall call t and t' semantically equivalent. Details follow. (We have made here the first step towards the extensive lattice-theoretic treatment in the more advanced theory of SCOTT, see e.g. [18]. The development of this is not necessary for our present purpose.)

As the next step, we extend the val function to terms τ and T. In this definition (and many of the subsequent formulations) we write $val(t,E)$, etc., instead of $val(t,J)$, etc., since it is only the E-component of J which interests us, the other components remaining fixed throughout.

DEFINITION 5.2.

1. $val(\tau,E)$ is defined as that function $f: V^n \to V$ which satisfies:
   $f(\bar{v}) = v$ iff, for $new\ \bar{x}$, $val(\tau(\bar{x}),E_{\bar{v}}^{\bar{x}}) = v$.

2. $val(T,E)$ is defined as that functional $F: (V^n \to V)^r \to (V^n \to V)$, which satisfies:
   $F(\bar{f}) = f$ iff, for $new\ \bar{\xi}$, $val(T(\bar{\xi}),E_{\bar{f}}^{\bar{\xi}}) = f$.

As the first lemma about the extended val we state

LEMMA 5.3.

1. *If none of the $\bar{z},\bar{u}$ occur free in s,*
   $val(\nu\bar{x}\lambda\bar{y}.s,E) = val(\nu\bar{z}\lambda\bar{u}.s[\bar{z}/\bar{x}][\bar{u}/\bar{y}],E)$.

2. *If none of the $\bar{\eta}$ occur free in σ,*
   $val(\lambda\bar{\xi}.\sigma,E) = val(\lambda\bar{\eta}.\sigma[\bar{\eta}/\bar{\xi}],E)$.

PROOF. Follows from the definitions and lemma 3.3. $\square$

REMARK. This lemma is, clearly, the analogue of the rule of α-conversion in the ordinary lambda calculus. It allows us a rewriting of bound variables where this is convenient.

The next lemma is of considerable technical importance in our develop-

ment. It is the main tool in the proof of the monotonicity theorem of the next section, which, in turn, plays an important part in the proof of the least fixed point theorem.

LEMMA 5.4.

Assume

1. $val(t,E) = v$, $val(\tau,E) = f$

2.1. $val(s,E) \subseteq val(s',E')$

2.2. $val(\sigma,E) \subseteq val(\sigma',E')$

2.3. $val(S,E) \subseteq val(S',E')$

3.1. $val(s_i,E) \subseteq val(s_i',E')$, $i=1,\ldots,n$

3.2 $val(\sigma_j,E) \subseteq val(\sigma_j',E')$, $j=1,\ldots,r$

4. $v' \subseteq v''$, $f' \subseteq f''$.

Then

1.1. $val(s[t/x],E) = val(s,E_v^x)$

1.2. $val(\sigma[t/x],E) = val(\sigma,E_v^x)$

1.3. $val(S[t/x],E) = val(S,E_v^x)$

1.4. $val(s[\tau/\xi],E) = val(S,E_f^\xi)$

1.5. $val(\sigma[\tau/\xi],E) = val(\sigma,E_f^\xi)$

1.6. $val(S[\tau/\xi],E) = val(S,E_f^\xi)$

2.1. $val(\sigma(s_1,\ldots,s_n),E) \subseteq val(\sigma'(s_1',\ldots,s_n'),E')$

2.2. $val(S(\sigma_1,\ldots,\sigma_r),E) \subseteq val(S'(\sigma_1',\ldots,\sigma_r'),E')$

3.1. $val(s,E_{v'f'}^{x\,\xi}) \subseteq val(s,E_{v''f''}^{x\,\xi})$

3.2. $val(\sigma,E_{v'f'}^{x\,\xi}) \subseteq val(\sigma,E_{v''f''}^{x\,\xi})$

3.3. $val(S,E_{v'f'}^{x\,\xi}) \subseteq val(S,E_{v''f''}^{x\,\xi})$

PROOF. By simultaneous induction on the complexity of the terms. We prove a few selected cases:

r

1.1. $s \equiv a$: $val(a[t/x],E) = val(a,E) = val(a,E_v^x)$.

$s \equiv x$: $val(x[t/x],E) = val(t,E) = v = val(x,E_v^x)$.

$s \equiv y \neq x$: $val(y[t/x],E) = val(y,E) = val(y,E_v^x)$.

$s \equiv \sigma(\bar{s})$: $val(\sigma(\bar{s})[t/x],E) = val(\sigma[t/x](\bar{s}[t/x]),E)$.

We have, by conclusions 1.2, 1.1, and induction:

$$val(\sigma[t/x],E) = val(\sigma,E_v^x)$$

$$val(\bar{s}[t/x],E) = val(\bar{s},E_v^x)$$

Hence, by conclusion 2.1 applied twice, and induction,

$$val(\sigma[t/x](\bar{s}[t/x]),E) = val(\sigma(\bar{s}),E_v^x)$$

$s \equiv \underline{if}\ p\ \underline{then}\ s'\ \underline{else}\ s''$. For the reader.

1.2. The cases that σ is a constant, variable or application are clear. Now let $\sigma \equiv \nu\bar{x}\lambda\bar{y}.s$. By suitably rewriting of bound variables (lemma 5.3) we may assume that none of the $\bar{x}$ or $\bar{y}$ occurs free in s or t. Then $val((\nu\bar{x}\lambda\bar{y}.s)[t/x],E) = val(\nu\bar{x}\lambda\bar{y}.s[t/x],E)$. We have, by definition 5.2, $val(\nu\bar{x}\lambda\bar{y}.s[t/x],E) = val(\nu\bar{x}\lambda\bar{y}.s,E_v^x)$ iff, for new $\bar{z}$ and arbitrary $\bar{w}$, $val((\nu\bar{x}\lambda\bar{y}.s[t/x])(\bar{z}),E_{\bar{w}}^{\bar{z}}) = val((\nu\bar{x}\lambda\bar{y}.s)(\bar{z}),E_{v\bar{w}}^{x\bar{z}})$, or, by definition of val and since the $\bar{z}$ are new, $val(s[t/x][\bar{z}/\bar{y}],E_{\bar{w}}^{\bar{z}}) = val(s[\bar{z}/\bar{y}],E_{v\bar{w}}^{x\bar{z}})$ (we assume that none of the $w_h = \perp$; otherwise, the result is obvious) or, by lemma 3.3, $val(s[\bar{z}/\bar{y}][t/x],E_{\bar{w}}^{\bar{z}}) = val(s[\bar{z}/\bar{y}],E_{\bar{w}v}^{\bar{z}x})$, and this holds by conclusion 1.1 of the lemma and induction.

1.3, 1.4, 1.5, 1.6. For the reader

2.1. By assumption 2.2, $val(\sigma,E) \subseteq val(\sigma',E')$, i.e. by definition 5.2, $val(\sigma(\bar{x}),E_{\bar{v}}^{\bar{x}}) \subseteq val(\sigma(\bar{x}),E_{\bar{v}}^{\bar{x}})$, for new $\bar{x}$ and arbitrary $\bar{v}$. By conclusion 3.1 and induction, $val(\sigma'(\bar{x}),E'_{\bar{v}}^{\bar{x}}) \subseteq val(\sigma'(\bar{x}),E'_{\bar{v}'}^{\bar{x}})$, if $\bar{v} \subseteq \bar{v}'$. Now choose $\bar{v} = val(\bar{s},E)$, $\bar{v}' = val(\bar{s}',E')$. Then $\bar{v} \subseteq \bar{v}'$ by assumption 3.1. We then have

$val(\sigma(\bar{x}),E_{\bar{v}}^{\bar{x}}) \subseteq val(\sigma'(\bar{x}),E'_{\bar{v}'}^{\bar{x}})$ (derived above)

$val(\sigma(\bar{x})[\bar{s}/\bar{x}],E) = val(\sigma(\bar{x}),E_{\bar{v}}^{\bar{x}})$ (by conclusion 1.1 and induction)

$val(\sigma'(\bar{x})[\bar{s}'/\bar{x}],E') = val(\sigma'(\bar{x}),E'_{\bar{v}'}^{\bar{x}})$ (similarly)

Hence, $val(\sigma(\bar{s}),E) \subseteq val(\sigma'(\bar{s}'),E')$, as was to be shown.

2.2 Omitted.

3.1. The cases that s is a constant or variable are clear. Now let $s \equiv \sigma(\bar{s})$. We have $val(\sigma, E^{x}_{v'} {}^{\xi}_{f'}) \subseteq val(\sigma, E^{x}_{v''} {}^{\xi}_{f''})$, by conclusion 3.2 and induction, $val(\bar{s}, E^{x}_{v'} {}^{\xi}_{f'}) \subseteq val(\bar{s}, E^{x}_{v''} {}^{\xi}_{f''})$, by conclusion 3.1 and induction; the result then follows by two applications of conclusion 2.1 and induction. The case that s is a selection is, once more, for the reader.

3.2, 3.3. Omitted. □

As the first consequence of lemma 5.4 we have lemma 5.5, which shows that our extension of the definition of val as given in definition 5.1, is consistent.

LEMMA 5.5.
1. $val(\tau(\bar{t}), E) = val(\tau, E)(val(\bar{t}, E))$.
2. $val(T(\bar{\tau}), E) = val(T, E)(val(\bar{\tau}, E))$.

PROOF.
1. Let $val(\bar{t}, E) = \bar{v}$. Then $val(\tau, E)(val(\bar{t}, E)) = val(\tau, E)(\bar{v}) =$
= (df. 5.2)$val(\tau(\bar{x}), E^{\bar{x}}_{\bar{v}}) =$ (lemma 5.4, part 1.1)$val(\tau(\bar{x})[\bar{t}/\bar{x}], E) =$
= $val(\tau(\bar{t}), E)$.
2. Similar. □

We are also in the position to show that changing call-by-value to call-by-name for one or more of the parameters yields a possibly extended function:

LEMMA 5.6. $val(\nu x_1 \ldots x_{\ell+1} \lambda x_{\ell+2} \ldots x_n \cdot t, E) \subseteq val(\nu x_1 \ldots x_\ell \lambda x_{\ell+1} \ldots x_n \cdot t, E)$.

PROOF. We show that, for new $\bar{z} = (z_1, \ldots, z_n)$
$val((\nu x_1 \ldots x_{\ell+1} \lambda x_{\ell+2} \ldots x_n \cdot t)(\bar{z}), E^{\bar{z}}_{\bar{v}}) \subseteq val((\nu x_1 \ldots x_\ell \lambda x_{\ell+1} \ldots x_n \cdot t)(\bar{z}), E^{\bar{z}}_{\bar{v}})$.
If any of the z_n, $1 \leq h \leq \ell+1$, equals $\perp$, the left-hand side of this inclusion has $\perp$ as value, and we are done. Otherwise, since the $\bar{z}$ are new,
$val((\nu x_1 \ldots x_{\ell+1} \lambda x_{\ell+2} \ldots x_n \cdot t)(\bar{z}), E^{\bar{z}}_{\bar{v}}) =$ (def. 4.1)
$val(t[z_h/x_h]^{\ell+1}_{h=1}[z_k/x_k]^n_{k=\ell+2}, E^{\bar{z}}_{\bar{v}}) = val(t[z_h/x_h]^\ell_{h=1}[z_k/x_k]^n_{k=\ell+1}, E^{\bar{z}}_{\bar{v}}) =$
$val((\nu x_1 \ldots x_\ell \lambda x_{\ell+1} \ldots x_n \cdot t)(\bar{z}), E^{\bar{z}}_{\bar{v}})$. □

We can now, at last, introduce the formalism in which we shall below state the main theorem of our paper.

We are interested in proving *assertions* about *formulae* $\Phi, \Psi \ldots$. A formulae is a *set* of *atomic* formulae, and an atomic formula is an inclusion of one of the forms $t_1 \subseteq t_2$, $\tau_1 \subseteq \tau_2$, or $T_1 \subseteq T_2$. An assertion has the form

$$\Phi \models_{\mathcal{D}} \Psi$$

For assertions in this format we introduce the notion of *validity* in

DEFINITION 5.6 (Validity of assertions).

Let $\mathcal{D}$ be a given mapping from (boolean) procedure symbols to (boolean) function terms.

1. Let $t_1 \subseteq t_2$ be an atomic formula. We call this formula *satisfied* by an interpretation J, iff $val(t_1, J) \subseteq val(t_2, J)$. Similarly for $\tau_1 \subseteq \tau_2$ and $T_1 \subseteq T_2$.

2. J satisfies a formula Φ iff it satisfies each element of the set Φ. If $J = <V,C,E;\mathcal{D}>$ satisfies Φ, we also say that "$\Phi[V,C,E;\mathcal{D}]$ holds".

3. An assertion $\Phi \models_{\mathcal{D}} \psi$ is called *valid* iff:
   For all V,C, whenever, for all E, $\Phi[V,C,E;\mathcal{D}]$ holds, then, for all E, $\Psi[V,C,E;\mathcal{D}]$ holds.

Note carefully the structure of clause 3 in the definition. Firstly, $\mathcal{D}$ will remain fixed in the application we have in mind, and is not subject to quantification. However, we emphasize the difference between the role of the V and C on the one hand, and E on the other hand: We define ... $\models$ ... as a statement of the form $\forall V, C[\forall E \ldots \supset \forall E \ldots]$, and not of the form $\forall V, C, E[\ldots \supset \ldots]$. In order to explain this, consider for instance the desired monotonicity property, which includes as special case:
$\{x \subseteq y\} \models_{\mathcal{D}} \{x[t/x] \subseteq y[t/x]\}$. Now, according to the second (rejected) definition, this means that for all E, if $E(x) \subseteq E(y)$ then $val(t,E) \subseteq E(y)$, which is clearly absurd. According to the first (adopted) definition, all it states is that, if for all E, $E(x) \subseteq E(y)$, then for all E, $val(t,E) \subseteq E(y)$, and this application does hold since its antecedent is false.

The section is concluded with some additional pieces of notation:

1. For $\Phi \models_{\mathcal{D}} \Psi$ we write, in the case that $\mathcal{D}$ is understood, just $\Phi \models \Psi$.

2. For $\phi \models \Psi$, with ϕ the empty set, we write $\models \Psi$.

3. When confusion is improbable, we omit the $\{\}$ around a collection $\Phi = \{\rho_1, \rho_2, \ldots\}$ of atomic formulae.

4. For $\models t_1 \subseteq t_2$, $t_2 \subseteq t_1$ we write $\models t_1 = t_2$, and similarly with $\models \tau_1 = \tau_2$ and $\models T_1 = T_2$.

6. MONOTONICITY

The first theorem of the paper states the monotonicity of our terms: The syntactical constructions of substitution, application, abstraction and selection all preserve the semantic ordering "$\subseteq$" between terms. (Recursion also preserves "$\subseteq$", but this can be proved only *after* the least fixed point result has been established. No further attention will be paid to this; the reader will have no problem to adapt the proof e.g. in [1] to the present formalism.)

THEOREM 6.1. (Monotonicity).

$s \subseteq s'$, $t \subseteq t'$, $t_i \subseteq t_i'$, $i=1,\ldots,n$, $p \subseteq p'$

$\sigma \subseteq \sigma'$, $\tau \subseteq \tau'$, $\tau_j \subseteq \tau_j'$, $j=1,\ldots,r$

$S \subseteq S'$,

$\models$

$s[t/x] \subseteq s'[t'/x]$,

$\sigma[t/x] \subseteq \sigma'[t'/x]$,   (Substitution,1)

$S[t/x] \subseteq S'[t'/x]$,

$s[\tau/\xi] \subseteq s'[\tau'/\xi]$,

$\sigma[\tau/\xi] \subseteq \sigma'[\tau'/\xi]$,   (Substitution,2)

$S[\tau/\xi] \subseteq S'[\tau'/\xi]$,

$\sigma(t_1,\ldots,t_n) \subseteq \sigma'(t_1',\ldots,t_n')$,

$S(\tau_1,\ldots,\tau_r) \subseteq S'(\tau_1',\ldots,\tau_r')$,   (Application)

$$\nu x_1 \ldots x_\ell \lambda y_1 \ldots y_m . t \subseteq \nu x_1 \ldots x_\ell \lambda y_1 \ldots y_m . t' ,$$

$$\lambda \xi_1 \ldots \xi_r . \tau \subseteq \lambda \xi_1 \ldots \xi_r . \tau' ,$$

(Abstraction)

$\underline{if}$ p $\underline{then}$ s $\underline{else}$ t $\subseteq$

$\underline{if}$ p' $\underline{then}$ s' $\underline{else}$ t'.

(Selection)

PROOF.

1. (Substitution). $s[t/x] \subseteq s'[t'/x]$. Choose any E, and let $val(t,E) = v$, $val(t',E) = v'$. Then $val(s[t/x],E) = val(s,E_v^x) \subseteq val(s,E_{v'}^x) \subseteq val(s',E_{v'}^x) = val(s'[t'/x],E)$, by lemma 5.4, part 1.1, lemma 5.4, part 3.1, the assumption, and lemma 5.4, part 1.1. The other cases for substitution are similar.

2. (Application). Direct from lemma 5.4, part 2.1.

3. (Abstraction). Let $\bar{z}$ be *new* variables, and let $\bar{v}$ be an arbitrary element in V^n. We shall show $val((\nu\bar{x}\lambda\bar{y}.t)(\bar{z}),E_{\bar{v}}^{\bar{z}}) \subseteq val((\nu\bar{x}\lambda\bar{y}.t')(\bar{z}),E_{\bar{v}}^{\bar{z}})$. If any of the v_h, $1 \le h \le \ell$, equals $\perp$, the whole evaluation on the left-hand side yields $\perp$, and we are done. Otherwise, we argue as follows: we apply the definition of val, the fact that $val(\bar{z},E_{\bar{v}}^{\bar{z}}) = \bar{v}$, and the fact that the $\bar{z}$ do not occur in $\nu\bar{x}\lambda\bar{y}.t$ or $\nu\bar{x}\lambda\bar{y}.t'$, and obtain successively:

$val((\nu\bar{x}\lambda\bar{y}.t)(\bar{z}),E_{\bar{v}}^{\bar{z}}) = $ (def. 4.1)

$val(t[\bar{z}/\bar{x}][\bar{z}/\bar{y}],E_{\bar{v}}^{\bar{z}}) = $ (lemma 5.4, part 1.1)

$val(t,E_{\bar{v}\bar{v}\bar{v}}^{\bar{z}\bar{x}\bar{y}}) \subseteq $ (assumption)

$val(t',E_{\bar{v}\bar{v}\bar{v}}^{\bar{z}\bar{x}\bar{y}}) = \ldots = val((\nu\bar{x}\lambda\bar{y}.t')(\bar{z}),E_{\bar{v}}^{\bar{z}})$, as was to be shown. The functional case is left to the reader, as is

4. (Selection). $\square$

Statement and proof of the monotonicity theorem for the various boolean cases are omitted.

7. APPROXIMATIONS

We arrive at our last body of definitions and preparatory lemmas. We assume ·from now on that we deal with one fixed $\mathcal{D}$, defined on each of

$\phi_1, \ldots, \phi_r$, with $\mathcal{D}(\phi_j) \equiv \tau_j$, $j=1,\ldots,r$. The proof to be given presently has to have available terms which, for all interpretations, have non-terminating evaluations. This is the reason for the following conventions: We extend our set of procedure symbols with the symbol ϕ_0, with declaration $\mathcal{D}(\phi_0) \equiv \phi_0$. Hence, ϕ_0 is a procedure which, when evaluated, causes nothing but a call upon itself. Clearly, therefore, the evaluation of ϕ_0 terminates for no argument.

Let us write ω, Ω, O for the "nowhere defined" individual-, function- and functional terms defined by:

$$\omega \equiv \phi_0(\bar{z}), \text{ with } \bar{z} \text{ new variables,}$$

$$\Omega \equiv \nu\bar{x}\lambda\bar{y}.\omega,$$

$$O \equiv \lambda\bar{\xi}.\Omega.$$

It is left to the reader to verify that, for all J, $val(\omega,J) = \bot$, $val(\Omega,J) = f_0$, where $f_0(\bar{v}) = \bot$ for all $v \in V^n$, and $val(O,J) = F_0$, where $F_0(\bar{f}) = f_0$ for all $\bar{f} \in (V^n \to V)^r$.

The reason we are interested in these constructs is the following: We want to define a process of *approximation* to our terms. An intuitive explanation is given first. Consider an individual term t. In general, t contains one or more occurrences ("calls") of the (recursive) procedure symbols $\phi_1, \ldots, \phi_r$, and evaluation of t will result in a, more or less elaborate, "calling tree" for the ϕ, where, in general, some ϕ_{j1} may call a ϕ_{j2}, this calls ϕ_{j3}, etc., with the possibility that $\phi_{j_i} \equiv \phi_{j_k}$ for $i \neq k$. However complicated this process may be, we always have that, if the evaluation of t terminates with value $v \in V_0$, then the calling tree is finite. It is then possible to obtain the same v as value of a new term, which is derived from t by suitable *finite* replacement of procedure symbols by the bodies of their declarations, where the procedures at the innermost level are not called any more (in general as a result of selection choosing another branch). These innermost occurrences of procedure symbols may then be replaced by whatever term we like, without changing the outcome. We now choose for this the undefined term Ω just introduced, since this choice guarantees a convenient ordering of the approximations, as will be seen soon. To be somewhat more specific, we shall prove that, for each term t

and interpretation J, there exists a term $t^{(i)}$ (with i an integer which is derived from the size of the calling tree determined by t, and, therefore, depending upon J) such that $val(t,J) = val(t^{(i)},J)$, and, moreover, $t^{(i)}$ contains no occurrences of any procedure symbol.

The first step towards the precise formulation of this idea is the introduction of one more syntactic operation on terms t, τ and T. The operation is denoted by "$\sim$", and defined *with respect to* the collection $\{\phi_1,\ldots,\phi_r\}$. It amounts to the replacement, in the term at hand, of each occurrence of a procedure symbol ϕ_j by a *new* variable ξ_j, for $j=1,\ldots,r$. In other words:

a. $\tilde{x} \equiv x$, $\tilde{a} \equiv a$, $\tau(t)^\sim \equiv \tilde{\tau}(\tilde{t})$,

   $(\underline{if}\ p\ \underline{then}\ t'\ \underline{else}\ t'')^\sim \equiv \underline{if}\ \tilde{p}\ \underline{then}\ \tilde{t}'\ \underline{else}\ \tilde{t}''$.

b. $\tilde{\alpha} \equiv \alpha$, $\tilde{\xi} \equiv \xi$, $\tilde{\phi}_j \equiv \xi_j$, $T(\tau)^\sim \equiv \tilde{T}(\tilde{\tau})$,

   $(\nu\bar{x}\lambda\bar{y}.t)^\sim \equiv \nu\bar{x}\lambda\bar{y}.\tilde{t}$.

c. $(\lambda\bar{\xi}.\tau)^\sim \equiv \lambda\bar{\xi}.\tilde{\tau}$.

The approximations are now defined in

<u>DEFINITION</u> 7.1 (Approximations).

1. $t^{(0)} \equiv \omega$, $\tau^{(0)} \equiv \Omega$, $T^{(0)} \equiv 0$.

2. $t^{(i+1)} \equiv \tilde{t}[\tau_j^{(i)}/\xi_j]_{j=1}^r$, $i=0,1,\ldots$.

   $\tau^{(i+1)} \equiv \tilde{\tau}[\tau_j^{(i)}/\xi_j]_{j=1}^r$, $i=0,1,\ldots$

   $T^{(i+1)} \equiv \tilde{T}[\tau_j^{(i)}/\xi_j]_{j=1}^r$, $i=0,1,\ldots$

It should be observed here that the τ_j in this definition are the bodies of the procedures ϕ_j, and that the ξ_j are the new variables introduced in the definition of the $\sim$-operation. In words, the zero-th approximation to τ is Ω, the i+1-st approximation is a term resulting from τ by

- replacing all procedure symbols by new variables

- substituting for these new variables the i-th approximations to the τ_j

(This somewhat roundabout process is necessary since we cannot substitute for procedure symbols.)

   As first lemma on these new constructs we have

LEMMA 7.2.

1. $\phi_j^{(i+1)} \equiv \tau_j^{(i)} \ (\equiv (\mathcal{D}(\phi_j))^{(i)})$, $j=1,\ldots,r$, $i=0,1,\ldots$

2. $\models \tau = (\lambda\bar{\xi}.\tilde{\tau})(\phi_1,\ldots,\phi_r)$

3. $\models t^{(i)} \subseteq t$, $\tau^{(i)} \subseteq \tau$, $T^{(i)} \subseteq T$, $i=0,1,\ldots$

4. $s \subseteq t$, $\sigma \subseteq \tau$, $S \subseteq T$

   $\models$

   $s^{(i)} \subseteq t^{(i)}$, $\sigma^{(i)} \subseteq \tau^{(i)}$, $S^{(i)} \subseteq T^{(i)}$, $i=0,1,\ldots$

PROOF. Straightforward from the definitions. $\square$

The next lemma is the key result for the proof of the least fixed point theorem.

LEMMA 7.3. *For all* t *and* J, *if* $val(t,J) = v$, *then there exists some* i *such that* $val(t^{(i)},J) = v$.

PROOF. If $v = \bot$, take $i = 0$. Otherwise, $v \in V_0$, and $valt(t,J,N) = v$, for some N. We prove, by induction on N, the following statement: If $valt(t,J,N) = v$, then, for some i and M, $valt(t^{(i)},J,M) = v$.

1. $t \equiv a$ or $t \equiv x$. Take $i = 1$.
2. $t \equiv \tau(\bar{t})$.

   a. $\tau \equiv \phi_j$. We have successively

   $valt(\phi_j(\bar{t}),J,N) = v$, or, by definition 4.1,

   $valt(\tau_j(\bar{t}),J,N-1) = v$, or, by induction, for some i_0 and M,

   $valt((\tau_j(\bar{t}))^{(i_0)},J,M) = v$, or, by definition 3.3 and 7.1,

   $valt(\tau_j^{(i_0)}(\bar{t}^{(i_0)}),J,M) = v$, or, by lemma 7.2, part 1,

   $valt(\phi_j^{(i_0+1)}(\bar{t}^{(i_0)}),J,M) = v$, or, by monotonicity,

   $valt(\phi_j^{(i_0+1)}(\bar{t}^{(i_0+1)}),J,M) = v$, or, as above,

   $valt((\phi_j(\bar{t}))^{(i_0+1)},J,M) = v$.

   Taking $i = i_0+1$ thus proves this case.

   b. $\tau \equiv \alpha$. Let $valt(\alpha(\bar{t}),J,N) = v$. Then there exist v_k such that

   $valt(t_k,J,N_k) = v_k$, with $N_k < N$, for $k=1,\ldots,n$. Thus, by induction,

$valt(t_k^{(i_k)}, ,M_k) = v_k$. Now taking $i = \max(i_1,\dots,i_n)$ settles this case.

c. $\tau \equiv \xi$ or, $\tau \equiv T(\bar{\tau})$, or $\tau \equiv \nu\bar{x}\lambda\bar{y}.t$. These cases are proven similarly using induction and monotonicity.

3. $t \equiv \underline{if}\ p\ \underline{then}\ t'\ \underline{else}\ t''$. For the reader. $\square$

8. LEAST FIXED POINTS

The time for the payoff of our labour has arrived. We state and prove

THEOREM 8.1 (The least fixed point theorem).

Let $\phi_1,\dots,\phi_r$ *be procedure symbols with* $\mathcal{D}(\phi_j) \equiv \tau_j$, $j=1,\dots,r$. *Let us put* $T_j \equiv \lambda\bar{\xi}.\tilde{\tau}_j$. *Then*

1. $\models \{T_j(\phi_1,\dots,\phi_r) = \phi_j\}_{j=1}^{r}$

2. $\{T_j(\sigma_1,\dots,\sigma_r) = \sigma_j\}_{j=1}^{r} \models \{\phi_j \subseteq \sigma_j\}_{j=1}^{r}$

REMARK

1. The first statement tells us that the ϕ's *are* fixed points of the T's; the second that they are *least* w.r.t. "$\subseteq$".

2. Observe that the first statement is nothing but $\models \{\tau_j = \phi_j\}_{j=1}^{r}$. However, this formulation does not bring out the fixed point aspect, and we have taken no inconsiderable trouble to provide a notation – with all the extra's for functionals – which does emphasize this.

PROOF.

1. We show that $\models \{\tau_j = \phi_j\}_{j=1}^{r}$.

   a. $\subseteq$: Let $val(\tau_j(\bar{z}),\mathcal{E}_{\bar{v}}^{\bar{x}}) = v$. If $v = \bot$, we are done. Otherwise, there exists N such that $valt(\tau_j(\bar{z}),\mathcal{E}_{\bar{v}}^{\bar{z}},N) = v$. Then, by definition 4.1, $valt(\phi_j(\bar{z}),\mathcal{E}_{\bar{v}}^{\bar{z}},\ N+1) = v$, hence $val(\phi_j(\bar{z}),\mathcal{E}_{\bar{v}}^{\bar{z}}) = v$. This proves "$\subseteq$".

   b. $\supseteq$: Reverse the argument of part a.

2. We first show

(8.1) $\quad \{T_j(\sigma_1,\ldots,\sigma_r) \subseteq \sigma_j\}_{j=1}^r \models \{\phi_j^{(i)} \subseteq \sigma_j\}_{j=1}^r$, $\quad i=0,1,\ldots$

by induction on i.

a. $i = 0$. Immediate from the definitions.

b. Assume the result for some i:

(8.2) $\quad \{T_j(\sigma_1,\ldots,\sigma_r) \subseteq \sigma_j\}_{j=1}^r \models \{\phi_j^{(i)} \subseteq \sigma_j\}_{j=1}^r$.

By lemma 7.2 and the definitions,

$$\models \phi_j^{(i+1)} = \tau_j^{(i)}, \text{ and}$$

$$\models \tau_j^{(i)} = (T_j(\phi_1,\ldots,\phi_r))^{(i)} \quad (\equiv T_j^{(i)}(\bar{\phi}^{(i)})).$$

Now $T_j^{(i)} \equiv (\lambda\bar{\xi}.\tilde{\tau}_j)^{(i)} \equiv (\lambda\bar{\xi}.\tilde{\tau}_j)^{\sim}[\tau_j^{(i-1)}/\xi_j]_{j=1}^r \equiv (\lambda\bar{\xi}.\overset{\approx}{\tau})[\tau_j^{(i-1)}/\xi_j]_{j=1}^r \equiv$

$\equiv \lambda\bar{\xi}.\overset{\approx}{\tau} \equiv \lambda\bar{\xi}.\tilde{\tau} \equiv T_j$, as follows from the definitions of $"\sim"$ and of substitution. Thus we obtain

(8.3) $\quad \models \phi_j^{(i+1)} = T_j(\phi_1^{(i)},\ldots,\phi_r^{(i)})$.

By monotonicity we have

(8.4) $\quad \{\phi_j^{(i)} \subseteq \sigma_j\}_{j=1}^r \models \{T_j(\phi_1^{(i)},\ldots,\phi_r^{(i)}) \subseteq T_j(\sigma_1,\ldots,\sigma_r)\}_{j=1}^r$.

From (8.2), (8.3) and (8.4) we conclude

$$\{T_j(\sigma_1,\ldots,\sigma_r) \subseteq \sigma_j\}_{j=1}^r \models \{\phi_j^{(i+1)} \subseteq \sigma_j\}_{j=1}^r$$

thus completing the inductive proof of (8.1) (or, in fact, of a stronger version with "$\subseteq$" instead of "$=$" in its antecedent). Next, we choose some V and C, and assume that for all E, $val(T_j(\sigma_1,\ldots,\sigma_r),J) \subseteq val(\sigma_j,J)$, for $j=1,\ldots,r$ and with $J = \langle V,C,E;\mathcal{D}\rangle$. We show that then, for all E (using the notation with V and C suppressed), $val(\phi_j,E) \subseteq val(\sigma_j,E)$, i.e., for new $\bar{z}$ and arbitrary $\bar{v}$, $val(\phi_j(\bar{z}),E_{\bar{v}}^{\bar{z}}) \subseteq val(\sigma_j(\bar{z}),E_{\bar{v}}^{\bar{z}})$, $j=1,\ldots,r$. Assume

$val(\phi_j(\bar{z}),E_{\bar{v}}^{\bar{z}}) = v$. Then, by lemma 7.3, for some i, $val((\phi_j(\bar{z}))^{(i)},E_{\bar{v}}^{\bar{z}}) = $

$= v$, i.e., $val(\phi_j^{(i)}(\bar{z}),E_{\bar{v}}^{\bar{z}}) = v$. From (8.1) it then follows that

$val(\sigma_j(\bar{z}),E_{\bar{v}}^{\bar{z}}) = v$, as was to be shown. $\square$

We have completed the proof of the least fixed point theorem, thus achieving the goal of our paper.

REFERENCES

[1] DE BAKKER, J.W., *Recursive Procedures*, Mathematical Centre Tracts 24, Mathematisch Centrum, Amsterdam, 1971.

[2] DE BAKKER, J.W., *Fixed points in programming theory*, in: Foundations of Computer Science (J.W. de Bakker, ed.), p.1-49, Mathematical Centre Tracts 63, Mathematisch Centrum, 1974.

[3] DE BAKKER, J.W. & W.P. DE ROEVER, *A calculus for recursive program schemes*, in: Automata, Languages and Programming (M. Nivat, ed.), p.167-196, North-Holland, Amsterdam, 1973.

[4] DE BAKKER, J.W. &. L.G.L.T. MEERTENS, *On the completeness of the inductive assertion method*, to appear in J. of Comp. Syst. Sciences.

[5] HINDLEY, J.R., B. LERCHER & J.P. SELDIN, *Introduction to Combinatory Logic*, Cambridge University Press, Cambridge, 1972.

[6] KLEENE, S.C., *Introduction to Metamathematics*, North-Holland, Amsterdam, 1952.

[7] LANDIN, P.J., *The mechanical evaluation of expressions*, Comp. J, 6 (1964), p.308-320.

[8] MANNA, Z., S. NESS & J. VUILLEMIN, *Inductive methods for proving properties of programs*, CACM, 16 (1973), p.491-502.

[9] MANNA, Z. & J. VUILLEMIN, *Fixpoint approach to the theory of computation*, CACM, 15 (1972), p.528-536.

[10] McCARTHY, J., *A basis for a mathematical theory of computation*, in: Computer Programming and Formal Systems (P. Braffort & D. Hirschberg, eds.), p.33-70, North-Holland, Amsterdam, 1963.

[11] MILNER, R., *Implementation and applications of Scott's logic for computable functions*, in: Proc. of an ACM Conference on Proving Assertions about Programs, p.1-6, ACM, 1972.

[12] MORRIS, J.H., *Lambda-calculus models of programming languages*, Ph.D Thesis, M.I.T., 1968.

[13] MORRIS, J.H., *Another recursion induction principle*, CACM, 14 (1971), p.351-354.

[14] NIVAT, M., *On the interpretation of recursive program schemes*, Report A74/09, Saarland University, Saarbrücken, 1974.

[15] PARK, D., *Fixpoint induction and proof of program semantics*, in: Machine Intelligence, Vol. 5 (B. Meltzer & D. Michie, eds.) p.59-78, Edinburgh University Press, 1970.

[16] DE ROEVER, W.P., *Recursion and parameter-mechanisms: an axiomatic approach*, in: Automata, Languages and Programming (J. Loeckx, ed.), p.34-65, Lecture Notes in Computer Science Vol. 14, Springer-Verlag, Berlin etc., 1974.

[17] ROSEN, B.K., *Tree-manipulating systems and Church-Rosser theorems*, J.ACM, 20 (1973), p.160-187.

[18] SCOTT, D., *Outline of a mathematical theory of computation*, in: Proc. of the Fourth Annual Princeton Conference on Information Sciences and Systems, p.169-176, Princeton, 1970.

[19] SCOTT, D. & J.W. DE BAKKER, *A theory of programs*, unpublished notes, IBM Seminar, Vienna, 1969.

[20] SCOTT, D. & C. STRACHEY, *Towards a mathematical semantics for computer languages*, in: Proc. of the Symposium on Computers and Automata (J. Fox, ed.), p.19-46, Polytechnic Inst. of Brooklyn, 1971.