Theoretical Computer Science 2 (1976) 155–181.
© North-Holland Publishing Company

LEAST FIXED POINTS REVISITED

J. W. DE BAKKER
Mathematical Centre, Amsterdam, The Netherlands

Communicated by M. Nivat
Received November 1974
Revised May 1975

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.

0. Notation

Section 2

$s, t, t_i, t', \ldots$	individual	terms
$\sigma, \tau, \tau_j, \tau', \ldots$	function	
$S, T, \ldots$	functional	
$p, p', \ldots$	boolean individual	terms
$\pi, \ldots$	boolean function	
$P, \ldots$	boolean functional	
$a \in A,\ \alpha \in \mathcal{A}$	constants	
$b \in B,\ \beta \in \mathcal{B}$		
$x, y, z, u \in X,\ \xi, \eta \in \mathcal{X}$	variables	
$q \in Q, \chi \in \mathcal{Q}$		
$\phi \in \mathcal{F},\ \psi \in \mathcal{G}$	procedure symbols	
$\tau(t_1, \ldots, t_n),\ \pi(t_1, \ldots, t_n)$	application	
$T(\tau_1, \ldots, \tau_r),\ P(\pi_1, \ldots, \pi_r)$		
$\nu x_1 \ldots x_l \lambda y_1 \ldots y_m . t$	abstraction	
$\nu x_1 \ldots x_l \lambda y_1 \ldots y_m . p$		
$\lambda \xi_1 \ldots \xi_r . \tau,\ \lambda \chi_1 \ldots \chi_r . \pi$		

if p then t′ else t″ *if p then p′ else p″*	selection
$n(\tau)$, $\langle n(T), r(T)\rangle$	rank
$1 \leq i \leq n$, $1 \leq j \leq r$, $1 \leq h \leq l$, $1 \leq k \leq m$	indices
$\mathbf{t}, \boldsymbol{\tau}, \mathbf{x}, \boldsymbol{\xi}, \ldots$	vector notation
$\equiv$	syntactic identity between terms

Section 3

$s[t/x], s[\tau/\xi], \ldots$	substitution
$\mathscr{E}^{x\xi}_{vf}, \mathscr{E}^{x}_{v}, \ldots$	change of environment

Section 4

$\mathscr{V} = \mathscr{V}_0 \cup \{\perp\}$	domain
$\mathscr{C}$	constant-interpretation
$\mathscr{E}$	variable-interpretation (environment)
$\mathscr{D}$	declarations
$\mathscr{I} = \langle \mathscr{V}, \mathscr{C}, \mathscr{E}; \mathscr{D}\rangle$	interpretation
$v, w \in \mathscr{V}, f \in \mathscr{V}^n \to \mathscr{V}$, $F \in (\mathscr{V}^n \to \mathscr{V})^r \to (\mathscr{V}^n \to \mathscr{V})$	elements of (higher) domains
$valt\,(t, \mathscr{I}, N)$ $val\,(t, \mathscr{I}), val\,(t, \mathscr{E})$	evaluation functions

Section 5

$v_1 \subseteq v_2, f_1 \subseteq f_2, F_1 \subseteq F_2$	partial orderings
$val\,(\tau, \mathscr{I}), val\,(T, \mathscr{I})$	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
$\Phi \models_{\mathscr{D}} \Psi$	assertion
$\Phi[\mathscr{V}, \mathscr{C}, \mathscr{E}; \mathscr{D}]$	$\langle \mathscr{V}, \mathscr{C}, \mathscr{E}; \mathscr{D}\rangle$ satisfies Φ
$\Phi \models \Psi, \models \Psi, \models t_1 = t_2, \ldots$	simplified forms of assertions

Section 7

ϕ_0	the nowhere defined procedure
ω	$\phi_0(z)$
Ω	$\nu x \lambda y.\omega$
$\mathcal{O}$	$\lambda\xi.\Omega$
$\tilde{t}, \tilde{\tau}, \tilde{T}$	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]. Moreover, their work drew attention to a problem which inspired the present paper. This problem is stated e.g. in Manna and Vuillemin [9, p. 529]: "Many programming languages use implementations (such as call-by-value) which do not necessarily lead to the least fixed point"; or in Manna et al. [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. The present author believes that it is advantageous to take a different approach to these matters: We view the main assertion of Morris to be that the function determined by a procedure with parameters called-by-value (f_v, say) may be properly included in the function determined by the same procedure with parameters called-by-name (f_n, say). (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.) However, it may well be that f_v and f_n, though different, are *both* least fixed points, albeit of different functionals. In order to explain this, consider the example of a recursive procedure (from Manna and Vuillemin [9]):

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

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:

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

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

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

Clearly, $f_v \subsetneq f_n$. Writing

$$\phi(x, y) \Leftarrow \Phi(\phi)(x, y) \tag{1.2}$$

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 $f_v \subsetneqq f_n$).

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. As we understand the literature, this result has been obtained before for call-by-name only, and by using quite different proof methods, viz. in Cadiou's Stanford Ph.D Thesis, in Rosen [17] and 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 the framework of axiomatized relations. An elementary exposition, which does not involve the somewhat advanced logical and algebraic tools of Nivat and Rosen, 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. (Observe, however, that our results imply that the rule is valid for (each combination of) call-by-value *and* call-by-name; this should be contrasted with the position taken in [8, 9].) 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 deal with an intricate issue which has led to somewhat diverging opinions.

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

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

(ii) 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 $\mathcal{V}$, map constants and variables to elements of $\mathcal{V}$ (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 $\mathcal{V}$ 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.

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:

(a) *application*, apply a function to one or more arguments;

(b) *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);

(c) *recursion*, a mechanism for "declaring" and "calling" (possibly recursive) procedures is introduced;

(d) *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*,

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

Fig. 1. Intended interpretation of terms.

functions and *functionals* as values. The evaluation mechanism is, of course, designed in such a way that it models program execution, insofar 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 $\mathcal{V}$, such that the terms of our language will, in this interpretation, have values according to Fig. 1.

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* (see Fig. 2).

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

Fig. 2. The initial classes of symbols.

The syntax of our language is given in Figs. 3 and 4 (explanatory remarks follow 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)	$\alpha \in \mathcal{A}$	$\xi \in \mathcal{X}$	$T(\tau_1, \ldots, \tau_r)$
π	$\beta \in \mathcal{B}$	$\chi \in \mathcal{Q}$	$P(\pi_1, \ldots, \pi_r)$
T (functional)	–	–	–
P	–	–	–

Fig. 3(a). Syntax, first part.

term\operation	*abstraction*	*selection*	*recursion*
t	–	*if* p *then* t' *else* t''	–
(individual)			
p	–	*if* p *then* p' *else* p''	–
τ	$\nu x_1 \ldots x_l \lambda y_1 \ldots y_m . t$	–	$\phi \in \mathcal{F}$
(function)	$(l + m > 0)$		
π	$\nu x_1 \ldots x_l \lambda y_1 \ldots y_m . p$	–	$\psi \in \mathcal{G}$
T	$\lambda \xi_1 \ldots \xi_r . \tau$	–	
(functional)	$(r > 0)$		
P	$\lambda \chi_1 \ldots \chi_r . \pi$	–	

Fig. 3(b). 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:

$$a,\ x,\ \phi(x_1, x_2),\ (\lambda x . x)(a),\ \textit{if } p \textit{ then } a \textit{ else } \phi(\alpha(x_1), \phi(x_1, x_2)).$$

(2) Function terms:

$$\alpha,\ \xi,\ (\lambda \xi . \alpha)(\phi),\ \nu x_1 x_2 .\ \textit{if } p \textit{ then } a \textit{ else } \phi(\alpha(x_1), \phi(x_1, x_2)),$$

$$\lambda y_1 y_2 .\ \textit{if } p \textit{ then } a \textit{ else } \phi(\alpha(y_1), \phi(y_1, y_2))$$

(we adopt here the obvious conventions for $l = 0$ or $m = 0$).

(3) Functional terms:

$$\lambda \xi . \nu x_1 x_2 .\ \textit{if } p \textit{ then } a \textit{ else } \xi(\alpha(x_1), \xi(x_1, x_2)),$$

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

(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 understanding the syntactic definitions in Figs. 3(a), 3(b).

(1) (Reading the tables). Consider, e.g., the first line (after t in Fig. 3(a)). This should be read as:

(a) Each individual constant or individual variable is an individual term.

(b) If τ is a function term, and each t_i, $i = 1, \ldots, n$, is an individual term, then $\tau(t_1, \ldots, t_n)$ 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 construct

$\tau \equiv \nu x_1 \ldots x_l \lambda y_1 \ldots y_m . t$. 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

(a) the $l \geq 0$ arguments (formal parameters) $x_1, \ldots, x_l$ *called-by-value*,

(b) the $m \geq 0$ arguments (formal parameters) $y_1, \ldots, y_m$ *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 precise in Section 4, with the "normal" λ-abstraction in the lambda calculus, where the conversion order is (very much) left open.) Again anticipating, call-by-value parameters will be evaluated by changing the environment, 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_h, $1 \leq h \leq l$, y_k, $1 \leq k \leq m$, 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 $[(\mathcal{V}^{n_1} \to \mathcal{V}) \times (\mathcal{V}^{n_2} \to \mathcal{V}) \times \ldots \times (\mathcal{V}^{n_r} \to \mathcal{V})] \to (\mathcal{V}^n \to \mathcal{V})$, 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 approach 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 $\langle n(T), r(T) \rangle$. The rank is initially given for the α, ξ and ϕ, and for the other constructs it is defined as follows

(a) if $\tau \equiv \nu x_1 \ldots x_l \lambda y_1 \ldots y_m . t$, then $n(\tau) = l + m$,

(b) if $\tau \equiv T(\tau_1, \ldots, \tau_r)$, and $n(\tau_i) = n$ $(i = 1, \ldots, r)$, then $n(\tau) = n$,

(c) if $T \equiv \lambda \xi_1 \ldots \xi_r . \tau$, and $n(\tau) = n$, then $n(T) = n$ and $r(T) = r$,

(d) similar definitions hold for the boolean case.

Furthermore, we require

(e) if $t \equiv \tau(t_1, \ldots, t_n)$, then $n(\tau) = n$,

(f) 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

(g) each τ of rank $n = n(\tau)$ is to be interpreted as a function of arity $n : \mathcal{V}^n \to \mathcal{V}$,

(h) each T of rank $\langle n, r \rangle$ is to be interpreted as a functional: $(\mathcal{V}^n \to \mathcal{V})^r \to (\mathcal{V}^n \to \mathcal{V})$,

(i) 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 ($\in \mathcal{V}$), and with application restricted to the one-argument case. For our present purpose, the syntax as in Figs. 3(a), (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 $\tau(t)$ for $\tau(t_1, \ldots, t_n)$, $T(\tau)$ for $T(\tau_1, \ldots, \tau_r)$, x for $(x_1, \ldots, x_l)$, 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 adapted 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

Definition 3.1 (Occurrences).

(1) x occurs in t iff

(1.1) $t \equiv x$,

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

(1.3) $t \equiv$ *if* p *then* t' *else* t'', and x occurs in p, t' or t''.

(2) x occurs in τ iff

(2.1) $\tau \equiv \nu x \lambda y . t$, and x occurs in t,

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

(3) x occurs in T iff

(3.1) $T \equiv \lambda \xi . \tau$, and x occurs in τ.

(4) ξ occurs in t iff

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

(4.2) $t \equiv$ *if* p *then* t' *else* t'', and ξ occurs in p, t' or t''.

(5) ξ occurs in τ iff

(5.1) $\tau \equiv \xi$,

(5.2) $\tau \equiv \nu \mathbf{x} \lambda \mathbf{y}.t$, and ξ occurs in t,

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

(6) ξ occurs in T iff

(6.1) $T \equiv \lambda \boldsymbol{\xi}.\tau$, and ξ occurs in τ.

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

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

Definition 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 $\nu x_1 \ldots x_l \lambda y_1 \ldots y_m . t$, with $x \equiv x_h$, for some h, $1 \leqslant h \leqslant l$, or $x \equiv y_k$, for some k, $1 \leqslant k \leqslant m$.

(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, $1 \leqslant j \leqslant r$.

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

Examples. (a) x occurs bound in $\lambda x.x$, free in $\alpha(x)$, and bound *and* free in $(\lambda x.x)(\alpha(x))$. (b) Both ξ and x occur bound in $\lambda \xi.\lambda x.\alpha_1((\lambda \eta.\xi)(\eta)(x))$, 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 $s[t/x], \ldots, P[t/x]$, $s[\tau/\xi], \ldots, P[\tau/\xi]$. 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) $a[t/x] \equiv a$, $x[t/x] \equiv t$, $y[t/x] \equiv y$ $(y \not\equiv x)$,

(1.2) $\sigma(s)[t/x] \equiv \sigma[t/x](s[t/x])$,

(1.3) (*if* p *then* s' *else* s'') $[t/x] \equiv$ *if* $p[t/x]$ *then* $s'[t/x]$ *else* $s''[t/x]$.

(2) $\sigma[t/x]$.

(2.1) $\alpha[t/x] \equiv \alpha$, $\xi[t/x] \equiv \xi$, $\phi[t/x] \equiv \phi$,

(2.2) $(\nu \mathbf{x} \lambda \mathbf{y}.s)[t/x] \equiv$

$\equiv \nu \mathbf{x} \lambda \mathbf{y}.s$, if $x \equiv x_h$, for some h, $1 \leqslant h \leqslant l$, or $x \equiv y_k$, for some k, $1 \leqslant k \leqslant m$; otherwise

$\equiv \nu \boldsymbol{x} \lambda \boldsymbol{y}.s[t/x]$, if none of the $\boldsymbol{x}$ or $\boldsymbol{y}$ occurs free in t; otherwise

$\equiv \nu \boldsymbol{z} \lambda \boldsymbol{u}.s[\boldsymbol{z}/\boldsymbol{x}][\boldsymbol{u}/\boldsymbol{y}][t/x]$ where the $\boldsymbol{z}, \boldsymbol{u}$ are *new* variables,

(2.3) $S(\boldsymbol{\sigma})[t/x] \equiv S[t/x](\boldsymbol{\sigma}[t/x])$.

(3) $S[t/x]$.

(3.1) $(\lambda \boldsymbol{\xi}.\tau)[t/x] \equiv \lambda \boldsymbol{\xi}.\tau[t/x]$.

(4) $s[\tau/\xi]$, $\sigma[\tau/\xi]$, $S[\tau/\xi]$. We only give the central cases, the remaining ones then follow as in (1)–(3).

(4.1) $\xi[\tau/\xi] \equiv \tau$, $\eta[\tau/\xi] \equiv \eta$ $(\eta \not\equiv \xi)$,

(4.2) $(\lambda \boldsymbol{\xi}.\sigma)[\tau/\xi] \equiv$

$\equiv \lambda \boldsymbol{\xi}.\sigma$, if $\xi \equiv \xi_j$, for some j, $1 \leqslant j \leqslant r$; otherwise

$\equiv \lambda \boldsymbol{\xi}.\sigma[\tau/\xi]$, if none of the $\boldsymbol{\xi}$ occurs free in τ; otherwise

$\equiv \lambda \boldsymbol{\eta}.\sigma[\boldsymbol{\eta}/\boldsymbol{\xi}][\tau/\xi]$ where the $\boldsymbol{\eta}$ 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 substitution.

(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, then* $s[y/x][t/y] \equiv s[t/x]$.

(2) *If $y \not\equiv x$, and y is not free in t'', then* $s[t'/y][t''/x] \equiv s[t''/x][t'[t''/x]/y]$.

Proof. See Hindley et al. [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, $\mathscr{E}$ be any function mapping arguments x to values v, and arguments ξ to values f. Then $\mathscr{E}^{x\xi}_{vf}$ is a function which satisfies:

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

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

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

Extension of the notation to the vector case: $\mathscr{E}^{\mathbf{x}\boldsymbol{\xi}}_{\mathbf{v}\mathbf{f}}$, and restriction of it to cases such as $\mathscr{E}^{x}_{v}$, should be clear. Observe that a notation such as $\mathscr{E}^{x\xi}_{vf}$ 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 $\mathscr{I} = \langle \mathscr{V}, \mathscr{C}, \mathscr{E}; \mathscr{D} \rangle$ has the following components:

(1) A domain (non-empty set) $\mathscr{V}$.

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

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

(4) $\mathscr{D}$ (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. *It is required that these τ (or π) do not contain free variables.* (This technical restriction could be avoided at the cost of a more elaborate formalism, viz. by syntactically including the decleration information within each term.)

We now discuss the way in which the interpretations $\mathscr{I}$ 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 $\langle t, \mathscr{I} \rangle$ and $\langle p, \mathscr{I} \rangle$ will yield a value in $\mathscr{V}$ 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 $\mathscr{V}$ is *extended* with one special element $\perp$, which is not an element of $\mathscr{V}$, and which stands for "undefined". From now on, $\mathscr{V}$ will refer to this extended set, and the subset of all "defined" elements of $\mathscr{V}$ will be called $\mathscr{V}_0$; i.e., we have $\mathscr{V} = \mathscr{V}_0 \cup \{\perp\}$. 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 $val(t, \mathcal{J})$ yielding $v \in \mathcal{V}$, according to the following scheme: First we introduce the *partial* function $valt(t, \mathcal{J}, N)$ (*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 $val(t, \mathcal{J})$ in terms of $valt(t, \mathcal{J}, N)$.

Definition 4.1 (Terminating evaluations). Let $\mathcal{J} = \langle \mathcal{V}, \mathcal{C}, \mathcal{E}; \mathcal{D}\rangle$ be an interpretation, and t an individual term. $valt(t, \mathcal{J}, N)$ is defined by the following inductive definition:

(1) $t \equiv a \in A$.
If $\mathcal{C}(a) = v \in \mathcal{V}_0$, then $valt(a, \mathcal{J}, 1) = v$.

(2) $t \equiv x \in X$.
If $\mathcal{E}(x) = v \in \mathcal{V}_0$, then $valt(x, \mathcal{J}, 1) = v$.

(3) $t \equiv \tau(t_1, \dots, t_n) \equiv \tau(\boldsymbol{t})$.

(3.1) $\tau \equiv \alpha \in \mathcal{A}$.
If $valt(t_i, \mathcal{J}, N_i) = v_i \in \mathcal{V}_0$, $i = 1, \dots, n$, and $\mathcal{C}(\alpha)(\boldsymbol{v}) = v \in \mathcal{V}_0$, then

$$valt\left(\alpha(\boldsymbol{t}), \mathcal{J}, \left(\sum_i N_i\right) + 1\right) = v.$$

(3.2) $\tau \equiv \xi \in \mathcal{X}$.
If $valt(t_i, \mathcal{J}, N_i) = v_i \in \mathcal{V}_0$, $i = 1, \dots, n$, and $\mathcal{E}(\xi)(\boldsymbol{v}) = v \in \mathcal{V}_0$, then

$$valt\left(\xi(\boldsymbol{t}), \mathcal{J}, \left(\sum_i N_i\right) + 1\right) = v.$$

(3.3) (The central case). $\tau \equiv \nu \boldsymbol{x} \lambda \boldsymbol{y} . t'$.
If $valt(t_h, \mathcal{J}, N_h) = v_h \in \mathcal{V}_0$, $h = 1, \dots, l$, and

$$valt\ (t'[z_h/x_h]_{h=1}^{l}[t_{l+k}/y_k]_{k=1}^{m}, \langle \mathcal{V}, \mathcal{C}, \mathcal{E}_{\boldsymbol{v}}^{\boldsymbol{z}}; \mathcal{D}\rangle, N) = v,$$

where the $\boldsymbol{z} = (z_1, \dots, z_l)$ are *new* variables, then

$$valt\left((\nu \boldsymbol{x} \lambda \boldsymbol{y} . t)(\boldsymbol{t'}), \mathcal{J}, \left(\sum_h N_h\right) + N\right) = v.$$

(3.4) $\tau \equiv T(\tau_1, \dots, \tau_r) = T(\boldsymbol{\tau})$, where $T \equiv \lambda \boldsymbol{\xi} . \tau_0$.
If $valt(\tau_0[\boldsymbol{\tau}/\boldsymbol{\xi}](\boldsymbol{t}), \mathcal{J}, N) = v$, then

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

(3.5) $\tau \equiv \phi \in \mathcal{F}$.
If $valt(\mathcal{D}(\phi)(\boldsymbol{t}), \mathcal{J}, N) = v$, then

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

(4) $t \equiv$ *if p then t' else t''*.
If $valt(p, \mathcal{J}, N) = 1$ and $valt(t', \mathcal{J}, N') = v$, then

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

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

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

(5) The definition of $valt(p, \mathcal{J}, N)$ is completely analogous to (1)–(4) and omitted.

Definition 4.2 (Evaluations).
(1) $val(t, \mathcal{J}) = v$ if there exists N such that $valt(t, \mathcal{J}, N) = v$.
(2) $val(t, \mathcal{J}) = \perp$, otherwise.
(3) Similarly for $val(p, \mathcal{J})$.

The following remarks have to be made on these definitions:

(1) Observe that, if $valt(t, \mathcal{J}, N) = v$, then $v \in \mathcal{V}_0$. 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 $\mathcal{V}_0$, and this property is inherited by the "non-terminal" cases.

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

(3) (Clauses (3.1), (3.2) of Definition 4.1). Let $t \equiv \tau(\boldsymbol{t})$ 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, $i = 1, \ldots, n$, be in $\mathcal{V}_0$. Observe that, otherwise, $val(t, \mathcal{J})$ will, by Definition 4.2, be set to $\perp$. The requirement that v_i be in $\mathcal{V}_0$ 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 $v \in \mathcal{V}_0$ fits in with our scheme that *valt* defines only terminating computations.

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

(a) The value parameters t_h, $h = 1, \ldots, l$, 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 enviroment 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 x to z is explained by the possibility that the x occur free in the t_{l+k}.

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

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

(6) (Clause (3.5) of Definition 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 Definition 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 lemmas.

Firstly, we introduce a partial ordering on our domains:

Definition 5.1.

(1) For $v_1, v_2 \in \mathcal{V}$,

$$v_1 \subseteq v_2 \quad \text{iff} \quad v_1 = \bot \quad \text{or} \quad v_1 = v_2.$$

(2) For $f_1, f_2 \in \mathcal{V}^n \to \mathcal{V}$,

$$f_1 \subseteq f_2 \quad \text{iff, for all} \quad \boldsymbol{v} \in \mathcal{V}^n, \quad f_1(\boldsymbol{v}) \subseteq f_2(\boldsymbol{v}).$$

(3) For $F_1, F_2 \in (\mathcal{V}^n \to \mathcal{V})^r \to (\mathcal{V}^n \to \mathcal{V})$,

$$F_1 \subseteq F_2 \quad \text{iff, for all} \quad \boldsymbol{f} \in (\mathcal{V}^n \to \mathcal{V})^r, \quad F_1(\boldsymbol{f}) \subseteq F_2(\boldsymbol{f}).$$

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 $\mathcal{V}$, we know what $val(t, \mathcal{J}) \subseteq val(t', \mathcal{J})$ means, viz. that, for this $\mathcal{J}$, either the evaluation of t does not terminate, or, t and t' have the same value in $\mathcal{V}_0$. If the inclusion and its reverse hold for all $\mathcal{J}$, 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 $val(t, \mathscr{E})$, etc., instead of $val(t, \mathcal{J})$, etc., since it is only the $\mathscr{E}$-component of $\mathcal{J}$ which interests us, the other components remaining fixed throughout.

Definition 5.2.

(1) $val(\tau, \mathscr{E})$ is defined as that function $f : \mathcal{V}^n \to \mathcal{V}$ which satisfies:

$$f(\boldsymbol{v}) = v \quad \text{iff, for } \textit{new } \boldsymbol{x}, \quad val(\tau(\boldsymbol{x}), \mathscr{E}^{\boldsymbol{x}}_{\boldsymbol{v}}) = v.$$

(2) $val(T, \mathscr{E})$ is defined as that functional $F : (\mathcal{V}^n \to \mathcal{V})^r \to (\mathcal{V}^n \to \mathcal{V})$, which satisfies:

$$F(f)=f \quad \text{iff, for } new\ \xi, \quad val(T(\xi), \mathscr{E}^{\xi}_{f})=f.$$

As the first lemma about the extended *val* we state

Lemma 5.3. (1) *If none of the* z, u *occur free in* s,

$$val(\nu x \lambda y . s, \mathscr{E}) = val(\nu z \lambda u . s[z/x][u/y], \mathscr{E}).$$

(2) *If none of the* η *occur free in* σ,

$$val(\lambda \xi . \sigma, \mathscr{E}) = val(\lambda \eta . \sigma[\eta/\xi], \mathscr{E}).$$

Proof. Follows from the definitions and Lemma 3.4. □

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 development. 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*

$$val(t, \mathscr{E}) = v, \qquad val(\tau, \mathscr{E}) = f, \tag{1}$$

$$val(\sigma, \mathscr{E}) \subseteq val(\sigma', \mathscr{E}'), \tag{2.1}$$

$$val(S, \mathscr{E}) \subseteq val(S', \mathscr{E}'), \tag{2.2}$$

$$val(s_i, \mathscr{E}) \subseteq val(s'_i, \mathscr{E}'), \qquad i = 1, \ldots, n, \tag{3.1}$$

$$val(\sigma_j, \mathscr{E}) \subseteq val(\sigma'_j, \mathscr{E}'), \qquad j = 1, \ldots, r, \tag{3.2}$$

$$v' \subseteq v'', \qquad f' \subseteq f''. \tag{4}$$

Then

$$val(s[t/x], \mathscr{E}] = val(s, \mathscr{E}^{x}_{v}), \tag{1.1}$$

$$val(\sigma[t/x], \mathscr{E}) = val(\sigma, \mathscr{E}^{x}_{v}), \tag{1.2}$$

$$val(S[t/x], \mathscr{E}) = val(S, \mathscr{E}^{x}_{v}), \tag{1.3}$$

$$val(s[\tau/\xi], \mathscr{E}) = val(s, \mathscr{E}^{\xi}_{f}), \tag{1.4}$$

$$val(\sigma[\tau/\xi], \mathscr{E}) = val(\sigma, \mathscr{E}^{\xi}_{f}), \tag{1.5}$$

$$val(S[\tau/\xi], \mathscr{E}) = val(S, \mathscr{E}^{\xi}_{f}), \tag{1.6}$$

$$val(\sigma(s_1, \ldots, s_n), \mathscr{E}) \subseteq val(\sigma'(s'_1, \ldots, s'_n), \mathscr{E}'), \tag{2.1}$$

$$val(S(\sigma_1,\ldots,\sigma_r),\mathscr{E})\subseteq val(S'(\sigma'_1,\ldots,\sigma'_r),\mathscr{E}'), \tag{2.2}$$

$$val(s,\mathscr{E}^{x\,\xi}_{v'f'})\subseteq val(s,\mathscr{E}^{x\,\xi}_{v''f''}), \tag{3.1}$$

$$val(\sigma,\mathscr{E}^{x\,\xi}_{v'f'})\subseteq val(\sigma,\mathscr{E}^{x\,\xi}_{v''f''}), \tag{3.2}$$

$$val(S,\mathscr{E}^{x\,\xi}_{v'f'})\subseteq val(S,\mathscr{E}^{x\,\xi}_{v''f''}), \tag{3.3}$$

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

(1.1) $s\equiv a$: $val(a[t/x],\mathscr{E})=val(a,\mathscr{E})=val(a,\mathscr{E}^{x}_{v})$.
$s\equiv x$: $val(x[t/x],\mathscr{E})=val(t,\mathscr{E})=v=val(x,\mathscr{E}^{x}_{v})$.
$s\equiv y\not\equiv x$: $val(y[t/x],\mathscr{E})=val(y,\mathscr{E})=val(y,\mathscr{E}^{x}_{v})$.
$s\equiv\sigma(s)$: $val(\sigma(s)[t/x],\mathscr{E})=val(\sigma[t/x](s[t/x]),\mathscr{E})$.
We have, by conclusions (1.2), (1.1), and induction

$$val(\sigma[t/x],\mathscr{E})=val(\sigma,\mathscr{E}^{x}_{v}),\qquad val(s[t/x],\mathscr{E})=val(s,\mathscr{E}^{x}_{v}).$$

Hence, by conclusion (2.1) applied twice, and induction,

$$val(\sigma[t/x](s[t/x]),\mathscr{E})=val(\sigma(s),\mathscr{E}^{x}_{v}),$$

$s\equiv$ *if p then s' else s"*. For the reader.

(1.2) The cases that σ is a constant, variable or application are clear. Now let $\sigma\equiv\nu x\lambda y.s$. By suitably rewriting of bound variables (Lemma 5.3) we may assume that none of the x or y occurs free in s or t. Then $val((\nu x\lambda y.s)[t/x],\mathscr{E})=val(\nu x\lambda y.s[t/x],\mathscr{E})$. We have, by Definition 5.2, $val(\nu x\lambda y.s[t/x],\mathscr{E})=val(\nu x\lambda y.s,\mathscr{E}^{x}_{v})$ iff, for new z and arbitrary w,

$$val((\nu x\lambda y.s[t/x])(z),\mathscr{E}^{z}_{w})=val((\nu x\lambda y.s)(z),\mathscr{E}^{xz}_{vw}),$$

or, by definition of val and since the z are new, $val(s[t/x][z/y],\mathscr{E}^{z}_{w})=val(s[z/y],\mathscr{E}^{xz}_{vw})$ (we assume that none of the $w_h=\perp$; otherwise, the result is obvious) or, by Lemma 3.4, $val(s[z/y][t/x],\mathscr{E}^{z}_{w})=val(s[z/y],\mathscr{E}^{zx}_{wv})$, 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 (2.1), $val(\sigma,\mathscr{E})\subseteq val(\sigma',\mathscr{E}')$, i.e. by Definition 5.2, $val(\sigma(x),\mathscr{E}^{x}_{v})\subseteq val(\sigma'(x),\mathscr{E}^{x}_{v})$, for new x and arbitrary v. By conclusion (3.1) and induction, $val(\sigma'(x),\mathscr{E}'^{x}_{v})\subseteq val(\sigma'(x),\mathscr{E}'^{x}_{v'})$, if $v\subseteq v'$. Now choose $v=val(s,\mathscr{E})$, $v'=val(s',\mathscr{E}')$. Then $v\subseteq v'$ by assumption (3.1). We then have

$$val(\sigma(x),\mathscr{E}^{x}_{v})\subseteq val(\sigma'(x),\mathscr{E}'^{x}_{v'})\qquad\text{(derived above)},$$

$$val(\sigma(x)[s/x],\mathscr{E})=val(\sigma(x),\mathscr{E}^{x}_{v})\qquad\text{(by conclusion (1.1) and induction)},$$

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

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

(2.2) Omitted.

(3.1) The cases that s is a constant or variable are clear. Now let $s \equiv \sigma(s)$. We have $val(\sigma, \mathscr{E}^{x\xi}_{v'f'}) \subseteq val(\sigma, \mathscr{E}^{x\xi}_{v''f''})$, by conclusion (3.2) and induction, $val(s, \mathscr{E}^{x\xi}_{v'f'}) \subseteq val(s, \mathscr{E}^{x\xi}_{v''f''})$, 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 5.2, is consistent.

Lemma 5.5.
(1) $val(\tau(t), \mathscr{E}) = val(\tau, \mathscr{E})(val(t, \mathscr{E}))$.
(2) $val(T(\boldsymbol{\tau}), \mathscr{E}) = val(T, \mathscr{E})(val(\boldsymbol{\tau}, \mathscr{E}))$.

Proof. (1) Let $val(\boldsymbol{t}, \mathscr{E}) = \boldsymbol{v}$. Then

$$val(\tau, \mathscr{E})(val(\boldsymbol{t}, \mathscr{E})) = val(\tau, \mathscr{E})(\boldsymbol{v})$$
$$= (\text{Def. 5.2}) val(\tau(\boldsymbol{x}), \mathscr{E}^{\boldsymbol{x}}_{\boldsymbol{v}}) = (\text{Lemma 5.4, part (1.1)}) val(\tau(\boldsymbol{x})[\boldsymbol{t}/\boldsymbol{x}], \mathscr{E})$$
$$= val(\tau(\boldsymbol{t}), \mathscr{E}).$$

(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. $val(\nu x_1 \ldots x_{l+1} \lambda x_{l+2} \ldots x_n . t, \mathscr{E}) \subseteq val(\nu x_1 \ldots x_l \lambda x_{l+1} \ldots x_n . t, \mathscr{E})$.

Proof. We show that, for new $\boldsymbol{z} = (z_1, \ldots, z_n)$,

$$val((\nu x_1 \ldots x_{l+1} \lambda x_{l+2} \ldots x_n . t)(\boldsymbol{z}), \mathscr{E}^{\boldsymbol{z}}_{\boldsymbol{v}}) \subseteq val((\nu x_1 \ldots x_l \lambda x_{l+1} \ldots x_n . t)(\boldsymbol{z}), \mathscr{E}^{\boldsymbol{z}}_{\boldsymbol{v}}).$$

If any of the v_h, $1 \leqslant h \leqslant l+1$, equals $\bot$, the left-hand side of this inclusion has $\bot$ as value, and we are done. Otherwise, since the $\boldsymbol{z}$ are new,

$$val((\nu x_1 \ldots x_{l+1} \lambda x_{l+2} \ldots x_n . t)(\boldsymbol{z}), \mathscr{E}^{\boldsymbol{z}}_{\boldsymbol{v}}) = (\text{Def. 4.1}),$$
$$val(t[z_h/x_h]_{h=1}^{l+1}[z_k/x_k]_{k=l+2}^{n}, \mathscr{E}^{\boldsymbol{z}}_{\boldsymbol{v}}) = val(t[z_h/x_h]_{h=1}^{l}[z_k/x_k]_{k=l+1}^{n}, \mathscr{E}^{\boldsymbol{z}}_{\boldsymbol{v}})$$
$$= val((\nu x_1 \ldots x_l \lambda x_{l+1} \ldots x_n . t)(\boldsymbol{z}), \mathscr{E}^{\boldsymbol{z}}_{\boldsymbol{v}}). \quad \square$$

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

$\Phi \models_{\mathscr{D}} \Psi$.

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

Definition 5.7 (Validity of assertions). Let $\mathscr{D}$ 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 $\mathscr{J}$, iff $val(t_1, \mathscr{J}) \subseteq val(t_2, \mathscr{J})$. Similarly for $\tau_1 \subseteq \tau_2$ and $T_1 \subseteq T_2$.

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

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

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

The section is concluded with some additional pieces of notation:

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

(2) For $\emptyset \models \Psi$, with $\emptyset$ 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).

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

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

$$S \subseteq S',$$

$$\models$$

$$s[t/x] \subseteq s'[t'/x],$$
$$\sigma[t/x] \subseteq \sigma'[t'/x], \qquad \text{(Substitution, 1)}$$
$$S[t/x] \subseteq S'[t'/x],$$

$$s[\tau/\xi] \subseteq s'[\tau'/\xi],$$
$$\sigma[\tau/\xi] \subseteq \sigma'[\tau'/\xi], \qquad \text{(Substitution, 2)}$$
$$S[\tau/\xi] \subseteq S'[\tau'/\xi],$$

$$\sigma(t_1, \ldots, t_n) \subseteq \sigma'(t_1', \ldots, t_n'),$$
$$\qquad \text{(Application)}$$
$$S(\tau_1, \ldots, \tau_r) \subseteq S'(\tau_1', \ldots, \tau_r'),$$

$$\nu x_1 \ldots x_l \lambda y_1 \ldots y_m . t \subseteq \nu x_1 \ldots x_l \lambda y_1 \ldots y_m . t',$$
$$\qquad \text{(Abstraction)}$$
$$\lambda \xi_1 \ldots \xi_r . \tau \subseteq \lambda \xi_1 \ldots \xi_r . \tau',$$

if p then s else t $\subseteq$

if p' then s' else t'. (Selection)

Proof.

(1) (Substitution). $s[t/x] \subseteq s'[t'/x]$. Choose any $\mathscr{E}$, and let $val(t, \mathscr{E}) = v$, $val(t', \mathscr{E}) = v'$. Then $val(s[t/x], \mathscr{E}) = val(s, \mathscr{E}_v^x) \subseteq val(s, \mathscr{E}_{v'}^x) \subseteq val(s', \mathscr{E}_{v'}^x) = val(s'[t'/x], \mathscr{E})$, 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 $\boldsymbol{z}$ be *new* variables, and let $\boldsymbol{v}$ be an arbitrary element in $\mathscr{V}^n$. We shall show $val((\nu \boldsymbol{x} \lambda \boldsymbol{y} . t)(\boldsymbol{z}), \mathscr{E}_{\boldsymbol{v}}^{\boldsymbol{z}}) \subseteq val((\nu \boldsymbol{x} \lambda \boldsymbol{y} . t')(\boldsymbol{z}), \mathscr{E}_{\boldsymbol{v}}^{\boldsymbol{z}})$. If any of the v_h, $1 \leq h \leq l$, 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 $val(\boldsymbol{z}, \mathscr{E}_{\boldsymbol{v}}^{\boldsymbol{z}}) = \boldsymbol{v}$, and the fact that the $\boldsymbol{z}$ do not occur in $\nu \boldsymbol{x} \lambda \boldsymbol{y} . t$ or $\nu \boldsymbol{x} \lambda \boldsymbol{y} . t'$, and obtain successively:

$$val((\nu \boldsymbol{x} \lambda \boldsymbol{y} . t)(\boldsymbol{z}), \mathscr{E}_{\boldsymbol{v}}^{\boldsymbol{z}}) = \text{(Def. (4.1))},$$

$$val(t[z/x][z/y], \mathscr{E}^{z}_{v}) = \text{(Lemma 5.4, part (1.1))},$$

$$val(t, \mathscr{E}^{zxy}_{vvv}) \subseteq \text{(assumption)},$$

$$val(t', \mathscr{E}^{zxy}_{vvv}) = \ldots = val((\nu x\lambda y.t')(z), \mathscr{E}^{z}_{v}),$$

as was to be shown.

The functional case is left to the reader, as is (4) (Selection). □

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 $\mathscr{D}$, defined on each of $\phi_1, \ldots, \phi_r$, with $\mathscr{D}(\phi_j) \equiv \tau_j$, $j = 1, \ldots, r$. 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 $\mathscr{D}(\phi_0) \equiv \phi_0$. 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, $\mathcal{O}$ for the "nowhere defined" individual-, function- and functional terms defined by:

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

$$\Omega \equiv \nu x\lambda y.\omega,$$

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

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

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_{j_1} may call a ϕ_{j_2}, this calls ϕ_{j_3}, 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 $v \in \mathscr{V}_0$, 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 $\mathcal{J}$, 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 $\mathcal{J}$) such that $val(t, \mathcal{J}) = val(t^{(i)}, \mathcal{J})$, 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 $\{\phi_1, \dots, \phi_r\}$. 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 $j = 1, \dots, r$. In other words:

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

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

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

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

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

The approximations are now defined in

Definition 7.1 (Approximations).

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

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

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

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

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 0th approximation to τ is Ω, the $(i+1)$st approximation is a term resulting from τ by

(a) replacing all procedure symbols by new variables,

(b) substituting for these new variables the ith 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) $$\phi_j^{(i+1)} \equiv \tau_j^{(i)} (\equiv (\mathcal{D}(\phi_j))^{(i)}), \qquad j = 1, \dots, r, \qquad i = 0, 1, \dots .$$

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

$$\begin{array}{l} s \subseteq t, \quad \sigma \subseteq \tau, \quad S \subseteq T, \\ \models \\ s^{(i)} \subseteq t^{(i)}, \quad \sigma^{(i)} \subseteq \tau^{(i)}, \quad S^{(i)} \subseteq T^{(i)}, \qquad i = 0, 1, \ldots . \end{array}$$ (3)

Proof. Straightforward from the definitions. □

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

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

Proof. If $v = \perp$, take $i = 0$. Otherwise, $v \in \mathcal{V}_0$, and $valt(t, \mathcal{J}, N) = v$, for some N. We prove, by induction on N, the following statment: If $valt(t, \mathcal{J}, N) = v$, then, for some i and M, $valt(t^{(i)}, \mathcal{J}, M) = v$.

(1) $t \equiv a$ or $t \equiv x$. Take $i = 1$.

(2) $t \equiv \tau(t)$.

(a) $\tau \equiv \phi_j$. We have successively
$valt(\phi_j(t), \mathcal{J}, N) = v$, or, by Definition 4.1,
$valt(\tau_j(t), \mathcal{J}, N-1) = v$, or, by induction, for some i_0 and M,
$valt((\tau_j(t))^{(i_0)}, \mathcal{J}, M) = v$, or, by Definitions 3.3 and 7.1,
$valt(\tau_j^{(i_0)}(t^{(i_0)}), \mathcal{J}, M) = v$, or, by Lemma 7.2, part (1),
$valt(\phi_j^{(i_0+1)}(t^{(i_0)}), \mathcal{J}, M) = v$, or, by monotonicity,
$valt(\phi_j^{(i_0+1)}(t^{(i_0+1)}), \mathcal{J}, M) = v$, or, as above,
$valt((\phi_j(t))^{(i_0+1)}, \mathcal{J}, M) = v$.
Taking $i = i_0 + 1$ thus proves this case.

(b) $\tau \equiv \alpha$. Let $valt(\alpha(t), \mathcal{J}, N) = v$. Then there exist v_k such that $valt(t_k, \mathcal{J}, N_k) = v_k$, with $N_k < N$, for $k = 1, \ldots, n$. Thus, by induction, $valt(t_k^{(i_k)}, \mathcal{J}, M_k) = v_k$. Now taking $i = \max(i_1, \ldots, i_n)$ settles this case.

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

(3) $t \equiv$ *if p then t' else t''*. For the reader. □

Remark. As pointed out by the referee, it should be observed that this lemma — though sufficient for our present purposes — needs additional assumptions in order to be generalized to domains with a richer partial ordering than the one imposed on our $\mathcal{V}$ by Definition 5.1 (1).

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

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

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

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 extras for functions — which does emphasize this.

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

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

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

(2) We first show

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

by induction on i.

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

(b) Assume the result for some i:

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

By Lemma 7.2 and the definitions,

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

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

Now

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

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

as follows from the definitions of "$\sim$" and of substitution. Thus we obtain

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

By monotonicity we have

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

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). Next, we choose some $\mathscr{V}$ and $\mathscr{C}$, and assume that for all $\mathscr{E}$, $val(T_j(\sigma_1, \ldots, \sigma_r), \mathscr{J}) \subseteq val(\sigma_j, \mathscr{J})$, for $j = 1, \ldots, r$, and with $\mathscr{J} = \langle \mathscr{V}, \mathscr{C}, \mathscr{E}; \mathscr{D} \rangle$. We show that then, for all $\mathscr{E}$ (using the notation with $\mathscr{V}$ and $\mathscr{C}$ suppressed), $val(\phi_j, \mathscr{E}) \subseteq val(\sigma_j, \mathscr{E})$, i.e., for new z and arbitrary v, $val(\phi_j(z), \mathscr{E}_v^z) \subseteq val(\sigma_j(z), \mathscr{E}_v^z)$, $j = 1, \ldots, r$. Assume $val(\phi_j(z), \mathscr{E}_v^z) = v$. Then, by Lemma 7.3, for some i, $val((\phi_j(z))^{(i)}, \mathscr{E}_v^z) = v$, i.e., $val(\phi_j^{(i)}(z), \mathscr{E}_v^z) = v$. From (8.1) it then follows that $val(\sigma_j(z), \mathscr{E}_v^z) = v$, as was to be shown. $\square$

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

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. We also acknowledge critical comments from K. P. Apt and A. Nijholt.

References

[1] J. W. De Bakker, *Recursive Procedures*, Mathematical Centre Tracts **24** (Mathematisch Centrum, Amsterdam, 1971).

[2] J. W. De Bakker, Fixed points in programming theory, in: *Foundations of Computer Science*, J. W. de Bakker (ed.) Mathematical Centre Tracts **63** (Mathematisch Centrum, Amsterdam, 1975) 1–49.

[3] J. W. De Bakker and W. P. De Roever, A calculus for recursive program schemes, in: *Automata, Languages and Programming*, M. Nivat (ed.) (North-Holland, Amsterdam, 1973) 167–196.

[4] J. W. De Bakker and L. G. L. T. Meertens, On the completeness of the inductive assertion method, *J. Comput. System Sci.* **11** (1975) 323–357.

[5] J. R. Hindley, B. Lercher and J. P. Seldin, *Introduction to Combinatory Logic* (Cambridge Univ. Press, Cambridge, 1972).

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

[7] P. J. Landin, The mechanical evaluation of expressions, *Comput. J.* **6** (1964) 308–320.

[8] Z. Manna, S. Ness and J. Vuillemin, Inductive methods for proving properties of programs, *Comm. ACM* **16** (1973) 491–502.

[9] Z. Manna and J. Vuillemin, Fixpoint approach to the theory of computation, *Comm. ACM* **15** (1972) 528–536.

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

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

[12] J. H. Morris, Lambda-calculus models of programming languages, Ph.D Thesis, M.I.T., Cambridge, Mass. (1968).

[13] J. H. Morris, Another recursion induction principle, *Comm. ACM* **14** (1971) 351–354.

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

[15] D. Park, Fixpoint induction and proof of program semantics, in: *Machine Intelligence*, **5**, B. Meltzer and D. Michie (eds.) (Edinburgh Univ. Press, Edinburgh, 1970) 59–78.

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

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

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

[19] D. Scott and J. W. De Bakker, A theory of programs, unpublished notes, IBM Seminar, Vienna (1969).

[20] D. Scott and C. Strachey, Towards a mathematical semantics for computer languages, in: *Proc. of the Symposium on Computers and Automata*, J. Fox (ed.) (Polytechnic Inst. of Brooklyn, New York, 1971) 19–46.