IA

stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 29/75 JANUARI

J.W. DE BAKKER

FIXED POINT SEMANTICS AND DIJKSTRA'S FUNDAMENTAL INVARIANCE THEOREM

Prepublication

5751.846 2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

AMS (MOS) subject classification scheme (1970): 68A05

ACM -Computing Reviews- category: 5.24

FIXED POINT SEMANTICS AND DIJKSTRA'S FUNDAMENTAL INVARIANCE THEOREM *)

by

J.W. de Bakker

ABSTRACT

In a recent paper, DIJKSTRA introduces a view of programming language semantics based on the notion of "weakest precondition". He uses this to axiomatize certain programming concepts, and in the formulation of his "Fundamental Invariance Theorem for Recursive Procedures". We shall demonstrate how these ideas can be incorporated in the framework of Scott's theory of mathematical semantics. For this purpose, a presentation of this theory in sofar as it is concerned with recursion, is given. We show that Dijkstra's theorem is incorrect, and that a modified version of it is immediately obtained from the fixed point approach to recursion, in particular using Scott's induction rule.

KEYWORDS & PHRASES: Semantics, recursion, least fixed points, induction, axiomatic approach, weakest precondition.

---

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

CONTENTS

1. INTRODUCTION

In [5], DIJKSTRA introduces a view of programming language semantics based on the notion of "weakest precondition". This concept is intended as a tool in refining Hoare's approach to semantics, as first proposed in [6], in which various language constructs are characterized axiomatically in terms of the relationship between pre- and postconditions. First, DIJKSTRA applies his idea to an analysis of the programming concepts of assignment, sequential composition, and conditional statements. He then considers the important notion of recursion, the treatment of which culminates in his Fundamental Invariance Theorem for Recursive Procedures.

The purpose of the present paper is to give an explanation of Dijkstra's ideas in the framework of mathematical semantics (for a general overview see SCOTT & STRACHEY [13]), in particular using the so-called fixed point theory. This theory, which dates back to Kleene's first recursion theorem [7], was revived in papers by MORRIS [10], SCOTT & DE BAKKER [12], and PARK [11], and has since found many applications (see [1,3,4,8,9,11] and the references given there). The two main results of the fixed point approach are

- Each recursive procedure determines a function which is the *least fixed point* of a functional associated in a natural way with the body of its declaration. This result is proved by considering the function as the limit of a sequence of approximations to it.
- The functional satisfies a certain *continuity* property.

Moreover, these two results are of importance for the justification of a powerful proof technique, called Scott's induction rule.

It will turn out that it is not difficult to explain Dijkstra's Fundamental Theorem in the framework of the fixed point theory. We shall see that the theorem, as given in [5], is incorrect, but that a slightly modified version of it is immediately obtained by an application of Scott's induction.

In order to make the paper self-contained, we develop the main properties of recursive procedures in sections 2 and 3. Since the essential results stated here are by now well-known, we have omitted some of (the details in) the proofs. On the other hand, we have tried to structure the argument somewhat more carefully than elsewhere, this being motivated by certain doubts shed on the least fixed point results in [8,9]. (Another attempt at clarification of these matters was made in DE BAKKER [2], which, contrary to our topic here, deals with recursive procedures with -explicit- parameters called-by-value and/or -by-name. The present paper deals only with recursive procedures which have the state as -implicit- parameter which may be considered called -by-value, if one so desires.)

In section 2 we introduce a formal language in which to write abstract programs, or, to use the technical term, *program schemes*. Next, it is described how a program scheme can be *interpreted* yielding a state-transforming function. In the definition of this, an important role is played by the notion of *computation sequence*, which is intended to model the usual meaning of sequential composition, conditional statements and procedure execution (using the "copy rule", i.e., replacing the procedure identifier by the body of its declaration).

In section 3 we present the fundamental properties of recursive program schemes. With each scheme a *functional* is associated, i.e., a function which maps state-transforming functions to state-transforming functions. It is convenient, for this purpose, to have available the notion of *program scheme variable*, which is therefore incorporated in our definition of program schemes. Next, the two fundamental results mentioned above (least fixed point property and continuity) are derived, followed by statement and justification of Scott's rule (in a slightly more general version than the one used e.g. in [1,3,4]).

Section 4 contains our explanation of Dijkstra's semantics. His notion of weakest precondition is defined in our framework, and the basic properties and rules which he postulates about this notion all turn out to be provable, thus yielding support to our claim of having formalized his informal definition. The "Fundamental Theorem" is then quoted, reformulated to fit into our system, and shown to be incorrect in this form. Next, a modification is suggested which leads to a new version which is, as announced before, a simple instance of Scott's induction.

2. RECURSIVE PROGRAM SCHEMES

We introduce a class of formal constructs, called *program schemes*, which are, in general, intended as a tool for investigating properties of the control structure of programs, and, in the present paper, in particular to clarify Dijkstra's approach to semantics.

Program schemes are linguistic expressions, i.e., they are built up from certain classes of symbols by applying a number of construction rules. As symbol classes we have
- the class $\mathcal{A}$ of elementary action symbols $A, A_1, \ldots$
- the class $\mathcal{P}$ of procedure symbols $P, P_1, \ldots$
- the class $\mathcal{B}$ of boolean symbols $p, q, \ldots$

For technical reasons which will become clear as we go along with the development of the theory, we also introduce
- the class $\mathcal{X}$ of program scheme variables $X, X_1, \ldots, Y, \ldots$
- the empty action symbol Ω.

We now give the syntax of program schemes in

DEFINITION 2.1 (Syntax of program schemes).

a. Each elementary action symbol, each procedure symbol, each program scheme variable, and the empty action symbol, are program schemes.

b. If S_1, S_2 are program schemes, and p is a boolean symbol, then $(S_1;S_2)$ and $(\underline{\text{if}}\ p\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2)$ are program schemes.

Meaning will be provided to program schemes by providing them with an *interpretation*, intended to reflect the usual programming language semantics in sofar as concerned with the concepts of interest here - these being sequential composition, selection and recursion. Before doing this, we state two more definitions on the purely syntactical level.

DEFINITION 2.2 (Syntactic identity between program schemes).
"$S_1 \equiv S_2$" is used to denote that S_1 and S_2 are identical sequences of symbols.

DEFINITION 2.3 (Substitution in a program scheme).
Let S, T be program schemes and let X be a program scheme variable. The

result of substituting T for X in S, i.e., of replacing all occurrences of X in S by T, is denoted by S[T/X] and obtained by induction on the structure of S as follows:

structure of S	S[T/X]
A, P or Ω	A, P or Ω
Y	$\begin{cases} T & (X \equiv Y) \\ Y & (X \not\equiv Y) \end{cases}$
$(S_1;S_2)$	$(S_1[T/X];S_2[T/X])$
(<u>if</u> p <u>then</u> S_1 <u>else</u> S_2)	(<u>if</u> p <u>then</u> $S_1[T/X]$ <u>else</u> $S_2[T/X]$)

We now describe how to *interpret* a program scheme. An interpretation M for a program scheme S is a means for providing sufficient information to "execute" S. If one so desires, one may view a program scheme as an abstract program which, by making certain choices for the unspecified parts in the scheme determines a concrete program, i.e., a specific way of computing a state-transforming function. In this approach, an interpretation M has to provide a number of things:

a. First of all, the domain V of states x,y,... has to be given. The interpreted program scheme will determine a state-transformation which, due to the possibility of non-termination, will in general be *partial*. Let $V \to V$ be the class of all partial functions from V to V, including ϕ (the *empty*, i.e., the nowhere defined, function). We shall also need the class of all partial functions $V \to \{\text{true},\text{false}\}$, in order to interpret boolean symbols.
b. Let C be a mapping from symbols to partial functions defined as follows:
   - For each $A \in \mathcal{A}$, $C(A) \in V \to V$.
   - For each $p \in \mathcal{B}$, $C(p) \in V \to \{\text{true},\text{false}\}$.
   - $C(\Omega) = \phi$.
c. Let E be a mapping from program scheme variables to partial functions: For each $X \in \mathcal{X}$, $E(X) \in V \to V$.
d. We still have to provide a meaning to the procedure symbols. In order to do this, we need *declarations*. Hence, as fourth component of M we have

$\mathcal{D} = \{<P,T> \mid P \in \mathcal{P}\}$, i.e., each procedure symbol P is provided with a *procedure body* T in the set of declarations $\mathcal{D}$.

Thus, each interpretation M is given as a four-tuple $M = <V,C,E,\mathcal{D}>$.

Next, we define how the meaning given to the various symbols is used in defining the meaning of whole schemes, doing this in a way which reflects the standard rules of program execution. Central to this definition will be the definition of the concept of *computation sequence*. First, however, we insert a remark directed to the reader who views our concrete programs as objects which still hardly resemble actual programs as he encounters them in practice. It may be helpful to his intuition to add one level to the interpretation, viz., to interpret the elementary action symbols as assignment statements, and to analyze the state into its components, such that in a notation intended to be self-explanatory, e.g., $x = \begin{pmatrix}\ldots i \ldots j \ldots \\ \ldots 1 \ldots 2 \ldots\end{pmatrix}$ is changed by the elementary action i := i+j to state $y = \begin{pmatrix}\ldots i \ldots j \ldots \\ \ldots 3 \ldots 2 \ldots\end{pmatrix}$. However, for our investigation which concentrates upon the control structure of the program we abstract from refinements needed for a treatment of assignment.

As another notational convention, we shall, for F any element in $V \to V$, sometimes use "xFy" as synonymous with "y = F(x)".

DEFINITION 2.4 (Computation sequence).
A computation sequence with respect to an interpretation $M = <V,C,E,\mathcal{D}>$ is a sequence

(2.1) $$x_0 S_0 x_1 \ldots x_i S_i x_{i+1} S_{i+1} \ldots x_{n-1} S_{n-1} x_n$$

such that

1. each S_i is a program scheme, $i=0,\ldots,n-1$.
2. $x_i \in V$, $i=0,\ldots,n$.
3. For each $i=0,\ldots,n-1$,
   a. if $S_i \equiv A$ or $S_i \equiv X$, for some $A \in \mathcal{A}$ or $X \in \mathcal{X}$, then $i = n-1$ and $x_{n-1} C(A) x_n$ or $x_{n-1} E(X) x_n$, respectively.
   b. If $S_i \equiv P$, for some $P \in \mathcal{P}$, then
      $x_{i+1} = x_i$,
      $S_{i+1} \equiv T$, where $<P,T> \in \mathcal{D}$.

c. If $S_i \equiv$ (<u>if</u> p <u>then</u> S' <u>else</u> S") then

$x_{i+1} = x_i$,

if $\mathcal{C}(p)(x_i)$ = true, then $S_{i+1} \equiv S'$, else if $\mathcal{C}(p)(x_i)$ = false, then $S_{i+1} \equiv S''$ (else, if $\mathcal{C}(p)(x_i)$ is undefined, then (2.1) is not a computation sequence).

d. If $S_i \equiv (S';S'')$, then, if

(i) $S' \equiv A$ or $S' \equiv X$, for some $A \in \mathcal{A}$ or $X \in \mathcal{X}$, then

$x_i \mathcal{C}(A) x_{i+1}$, or $x_i \mathcal{E}(X) x_{i+1}$, respectively,

$S_{i+1} \equiv S''$.

(ii) $S' \equiv P$, for some $P \in \mathcal{P}$, then

$x_{i+1} = x_i$,

$S_{i+1} \equiv (T;S'')$, where $\langle P,T\rangle \in \mathcal{D}$.

(iii) $S' \equiv$ (<u>if</u> p <u>then</u> S''' <u>else</u> S$^{\text{iv}}$), then

$x_{i+1} = x_i$,

$S_{i+1} \equiv$ (<u>if</u> p <u>then</u> (S''';S") <u>else</u> (S$^{\text{iv}}$;S"))

(iv) $S' \equiv (S''';S^{\text{iv}})$, then

$x_{i+1} = x_i$,

$S_{i+1} \equiv (S''';(S^{\text{iv}};S''))$.

4. $S_n \in \mathcal{A}$ or $S_n \in \mathcal{X}$.

We are now sufficiently prepared for

DEFINITION 2.5 (Semantics of program schemes).
Let S be a program scheme, and $M = \langle \mathcal{V},\mathcal{C},\mathcal{E},\mathcal{D}\rangle$ an interpretation. $M(S)$ is the function: $\mathcal{V} \to \mathcal{V}$, determined by: $M(S)(x) = y$ iff there exists a computation sequence $x_0S_0x_1\ldots x_{n-1}S_{n-1}x_n$ with respect to M, such that

a. $x_0 = x$, $x_n = y$.

b. $S_0 \equiv S$.

Our first lemma lists a number of basic properties which follow fairly directly from the definitions. Some of these properties are of the form

$$M(S_1) = M(S_2);$$

i.e., S_1 and S_2 are "equal under the interpretation M". As alternative notation for this we use

$$\models_M S_1 = S_2.$$

LEMMA 2.6. *For all* $M = \langle \mathcal{V},\mathcal{C},\mathcal{E},\mathcal{D}\rangle$:

a. $\models_M ((S_1;S_2);S_3) = (S_1;(S_2;S_3))$.

b. $\models_M P = T$, *where* $\langle P,T\rangle \in \mathcal{D}$.

c. $M(A) = \mathcal{C}(A)$
$M(X) = \mathcal{E}(X)$.

d. $\models_M ((\underline{\text{if}}\ p\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2);S) = (\underline{\text{if}}\ p\ \underline{\text{then}}\ (S_1;S)\ \underline{\text{else}}\ (S_2;S))$.

e. $M((S_1;S_2)) = M(S_1) \circ M(S_2)$
where "$\circ$" denotes functional composition (i.e., $(f\circ g)(x) = f(g(x))$).

PROOF. Not difficult, and left to the reader. □

REMARK. In the sequel, we shall enhance readability by omitting outermost parentheses around $(S_1;S_2)$ or $(\underline{\text{if}}\ p\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2)$.

The next lemma relates the operations of substitution and changing the interpretation. First another notation.

DEFINITION 2.7.
Let $M = \langle \mathcal{V},\mathcal{C},\mathcal{E},\mathcal{D}\rangle$, and let $F\colon \mathcal{V} \to \mathcal{V}$ be any function.
$M\{F/X\}$ is used to denote the new $\langle \mathcal{V}',\mathcal{C}',\mathcal{E}',\mathcal{D}'\rangle$ such that
a. $\mathcal{V}' = \mathcal{V}$, $\mathcal{C}' = \mathcal{C}$, $\mathcal{D}' = \mathcal{D}$.
b. $\mathcal{E}'(Y) = \mathcal{E}(Y)$ for all $Y \not\equiv X$,
$\mathcal{E}'(X) = F$.

In words, $M\{F/X\}$ is the same as M, apart from, possibly, the interpretation of X, which is now set to F. Clearly, $M\{F/X\}(X) = F$.

LEMMA 2.8.

$$M(T_1[T_2/X]) = M\{M(T_2)/X\}(T_1).$$

PROOF. Induction on the structure of T_1. □

3. FUNDAMENTAL PROPERTIES

First we introduce a partial ordering "$\subseteq$" on the elements of $V \to V$.

DEFINITION 3.1.
Let $F_1, F_2 \in V \to V$. $F_1 \subseteq F_2$ holds iff, for all $x,y \in V$, if $F_1(x) = y$, then $F_2(x) = y$.

REMARK. $F_1 \subseteq F_2$ holds iff, for all x, F_1 is either undefined in x, or, if $F_1(x)$ is defined, then $F_2(x)$ is also defined, and yields the same value as $F_1(x)$.

The next definition introduces the notion of a *chain* of partial functions, and of its least upper bound (lub). These notions play a part in the first two fundamental theorems of this section.

DEFINITION 3.2 (Chains and their lubs)
a. A *chain* is a sequence of partial functions $\{F_i\}_{i=0,1,\ldots}$ such that $F_i \subseteq F_{i+1}$, $i=0,1,\ldots$
b. For $\{F_i\}_{i=0,1,\ldots}$ a chain, we define $(\bigcup_{i=0}^{\infty} F_i)(x) = y$ iff $F_i(x) = y$, for some i.

LEMMA 3.3. *Let* $\{F_i\}_{i=0,1,\ldots}$ *be a chain. Then* $\bigcup_{i=0}^{\infty} F_i$ *is a function such that*
a. $F_i \subseteq \bigcup_{i=0}^{\infty} F_i$, $i=0,1,\ldots$
b. *If, for some* G, $F_i \subseteq G$, *for* $i=0,1,\ldots$, *then* $\bigcup_{i=0}^{\infty} F_i \subseteq G$,
i.e., $\bigcup_{i=0}^{\infty} F_i$ *is the* <u>*least*</u> <u>*upper*</u> <u>*bound*</u> *of the chain.*

PROOF. Straightforward from the definitions. □

We now extend our notation $\models_M S_1 = S_2$, as introduced in the previous section.

DEFINITION 3.4.
a. An *atomic formula* is an expression of the form $S_1 \subseteq S_2$ or $S_1 = S_2$. An atomic formula is *satisfied* by an interpretation M if $M(S_1) \subseteq M(S_2)$, or $M(S_1) = M(S_2)$, respectively, holds.

b. A *formula* (denoted by $\Phi,\Psi,\ldots$) is a set of zero or more atomic formulae. A formula is satisfied by an interpretation M if each of its elements is satisfied by M.

c. $\Phi \models_M \Psi$ holds iff the implication "if Φ satisfies M, then Ψ satisfies M" holds.

d. Empty Φ are omitted, as are brackets around sets of atomic formulae. (E.g., for $\phi \models_M \{S_1 \subseteq S_2, S_2 \subseteq S_1\}$ we write $\models_M S_1 \subseteq S_2,\ S_2 \subseteq S_1$ (which, clearly, holds iff $\models_M S_1 = S_2$ holds).)

LEMMA 3.5 (Monotonicity). *For each* M

a. *If* $F_1 \subseteq F_2$ *then, for all* S, $M\{F_1/X\}(S) \subseteq M\{F_2/X\}(S)$.

b. $T_1 \subseteq T_2 \models_M T[T_1/X] \subseteq T[T_2/X]$.

PROOF.

a. Formula induction on S.

b. By definition 3.4, part c, we have to show:
if $M(T_1) \subseteq M(T_2)$, then $M(T[T_1/X]) \subseteq M(T[T_2/X])$. By lemma 2.8, $M(T[T_i/X]) =$
$= M\{M(T_i)/X\}(T)$, $i=1,2$.
The result now follows by part a. □

The next step in our development is to associate with each program scheme S and interpretation M a functional σ_M, defined as follows:

DEFINITION 3.6. Let S be a program scheme, X a program scheme variable, and M an interpretation. The associated functional $\sigma_{M,X}$ (or σ_M, if X is understood), is defined as

$$\sigma_{M,X} = \lambda F \cdot M\{F/X\}(S).$$

In words, $\sigma_{M,X}$ is a mapping from $V \to V$ to $V \to V$ defined as follows: Each $F \in V \to V$ is mapped to the function which is obtained by applying the interpretation $M\{F/X\}$ to S, i.e., $\sigma_{M,X}(F) = M\{F/X\}(S)$, for each $F \in V \to V$.

We now present two lemma's which together establish that, for given $M = \langle V,C,E,\mathcal{D}\rangle$, if $\langle P,T\rangle \in \mathcal{D}$, then $M(P)$ is the least fixed point of the functional which is associated with T as described in definition 3.6, after one precaution is taken, as described in definition 3.7. (This precaution is necessary to ensure the availability of a variable in T. Since P is not itself a variable, one cannot directly use definition 3.6.)

DEFINITION 3.7. Let S be a program scheme, P a procedure symbol, and X a program scheme variable *not occurring in* S . S[P→X] denotes the result of replacing all occurrences of P in S by X, and is obtained by induction on the structure of S:

Structure of S'	S[P→X]
A, Y, or Ω	A, Y or Ω
P	X
P'(≠P)	P'
$S_1;S_2$	S_1[P→X];S_2[P→X]
<u>if</u> p <u>then</u> S_1 <u>else</u> S_2	<u>if</u> p <u>then</u> S_1[P→X] <u>else</u> S_2[P→X]

Clearly, S[P→X][P/X] ≡ S.

DEFINITION 3.8. Let M be an interpretation, let T be a program scheme, P a procedure symbol, and X a program scheme variable (not occurring in T). We define $\tau_{M,P,X}$ (or $\tau_{M,P}$, if X is understood) as

$$\tau_{M,P,X} = \lambda F \circ M\{F/X\}(T[P\to X]).$$

LEMMA 3.9. *Let, for any functional* σ, σ^i *be defined by* $\sigma^0(F) = F$, $\sigma^{i+1}(F) = \sigma(\sigma^i(F))$. *Let* $M = \langle V,C,E,D\rangle$ *be an interpretation, and let* P *be a procedure symbol with* $\langle P,T\rangle \in D$. *Then*

$$\bigcup_{i=0}^{\infty} \tau^i_{M,P}(\phi) \subseteq M(P).$$

PROOF. It is sufficient to show that $\tau^i_{M,P}(\phi) \subseteq M(P)$, for all i.

a. i = 0. $\phi \subseteq M(P)$, by definition of ϕ and "$\subseteq$".

b. Assume $\tau^i_{M,P}(\phi) \subseteq M(P)$. We show that, then, $\tau^{i+1}_{M,P}(\phi) \subseteq M(P)$:
$\tau^{i+1}_{M,P}(\phi) = \tau_{M,P}(\tau^i_{M,P}(\phi)) \subseteq$ (induction hypothesis and monotonicity) $\tau_{M,P}(M(P)) =$(df. $\tau_{M,P}$) $M\{M(P)/X\}(T[P\to X]) =$ ((lemma 2.8) $M(T[P\to X][P/X]) =$ = (remark after def. 3.7) $M(T)$ = ((lemma 2.6, part b) $M(P)$. □

LEMMA 3.10. *Let M, P, T and $\tau_{M,P}$ be as above. Then*

$$M(P) \subseteq \bigcup_{i=0}^{\infty} \tau_{M,P}^{i}(\phi).$$

PROOF. We prove a more general assertion: Let S be any program scheme, and let $\sigma_{M,P}$ be as in definition 3.8. Then

$$M(S) \subseteq \bigcup_{i=0}^{\infty} \sigma_{M,P}\left(\tau_{M,P}^{i}(\phi)\right).$$

It is sufficient to show: For each $x,y \in V$, if $x\, M(S)y$, then $x\ \sigma_{M,P}\left(\tau_{M,P}^{i}(\phi)\right)y$, for some i. Assume $x\, M(S)y$. By definition 2.5, there exists a computation sequence with respect to M

(3.1) $\qquad x_0S_0x_1\dots x_{n-1}S_{n-1}x_n$

such that $x = x_0$, $y = x_n$, and $S \equiv S_0$. We prove the assertion by induction on the entity (n,γ), where n is the number of applications of clause 2.4.3b or 2.4.3.d(ii) in the construction of the computation sequence (3.1) (i.e., the number of times that body replacement of T for P has taken place), and γ is the syntactic complexity of S_0 (a precise definition of which is left to the reader). As ordering on the (n,γ) we define $(n_1,\gamma_1) < (n_2,\gamma_2)$ iff $n_1 < n_2$, or $n_1 = n_2$, and $\gamma_1 < \gamma_2$. We have the following cases:

- $S \equiv A \in A$, or $P'(\neq P) \in P$, or $X \in X$.
  Then we may take $i = 0$. Indeed, we have that $M(S) \subseteq \sigma_{M,P}(\phi)$, since
  $$\begin{aligned}\sigma_{M,P}(\phi) &= [\lambda F\circ M\{F/X\}(S[P\to X])](\phi)\\ &= M\{\phi/X\}(S[P\to X])\\ &= M(S),\end{aligned}$$
  where the last equality holds since P does not occur in S, and, therefore, X not in $S[P\to X]$.
- $S \equiv P$. Let $x = x_0$, $y = x_n$, and let the computation sequence $x_0Px_1T\dots x_{n-1}S_{n-1}x_n$ have complexity (n,γ). By definition 2.4, part 3b, $x_1 = x_0$. Moreover, the computation sequence $x_1T\dots x_{n-1}S_{n-1}x_n$ has complexity $(n-1,\gamma') < (n,\gamma)$, where γ' is the syntactic complexity of T. By the induction hypothesis, applied with $S \equiv T$, and, hence, $\sigma_{M,P} = \tau_{M,P}$,

we have that x $\tau_{M,P}^{i_0}(\sigma_{M,P}(\phi))$y holds for some i_0. Thus x $\tau_{M,P}^{i_0+1}(\phi)$y holds, and we see that we can take for the i corresponding to S $\equiv$ P, i = i_0+1.

- S $\equiv S_1;S_2$. Let x M(S)y, with a computation sequence with complexity (n,γ). Since $M(S) = M(S_1)\circ M(S_2)$, there exists z such that x $M(S_1)$z, and z $M(S_2)$y, with complexity (n_1,γ_1) and (n_2,γ_2), respectively. Clearly, $n_1 \le n$, $\gamma_1 < \gamma$, $n_2 \le n$, $\gamma_2 < \gamma$. Hence, by the induction hypothesis, for some i_1,i_2, x $\sigma_{1M,P}(\tau_{M,P}^{i_1}(\phi))$z, and z $\sigma_{2M,P}(\tau_{M,P}^{i_2}(\phi))$y hold. Let i = max$(i_1,i_2)$. Then, by monotonicity (lemma 3.5), x $\sigma_{1M,P}(\tau_{M,P}^{i}(\phi))$z, and z $\sigma_{2M,P}(\tau_{M,P}^{i}(\phi))$y hold, and x $\sigma_{M,P}(\tau_{M,P}^{i}(\phi))$y follows, since $\sigma_{M,P}(\tau_{M,P}^{i}(\phi)) = \sigma_{1M,P}(\tau_{M,P}^{i}(\phi)) \circ \sigma_{2M,P}(\tau_{M,P}^{i}(\phi))$.
- S $\equiv$ <u>if</u> p <u>then</u> S_1 <u>else</u> S_2. Similar to the previous case, and left to the reader.

Thus, we have shown that $M(S) \subseteq \bigcup_{i=0}^{\infty} \sigma_{M,P}(\tau_{M,P}^{i}(\phi))$ holds for each S. Taking S $\equiv$ T, and applying lemma 2.6, part b, we obtain that $M(P) \subseteq \bigcup_{i=0}^{\infty} \tau_{M,P}^{i}(\phi)$, as was to be shown. □

THEOREM 3.11. *Let* P $\in$ P, M = $\langle V,C,E,D\rangle$ *with* $\langle$P,T$\rangle$ $\in$ D, *and let* $\tau_{M,P}$ *be as in definition* 3.8. *Then*

$$M(P) = \bigcup_{i=0}^{\infty} \tau_{M,P}^{i}(\phi).$$

PROOF. Lemma's 3.9 and 3.10. □

COROLLARY 3.12 (The least fixed point result).
Let M, P, T, $\tau_{M,P}$ *be as before. Then* M(P) *is the least fixed point of* $\tau_{M,P}$.

PROOF. By lemma 2.6, part b, M(P) *is* a fixed point of $\tau_{M,P}$. Now let F be any function such that $\tau_{M,P}(F) = F$. We show that then $M(P) \subseteq F$. By theorem 3.11, it is sufficient to show that $\tau_{M,P}^{i}(\phi) \subseteq F$, for each i. The case i = 0 is clear. Now assume $\tau_{M,P}^{i}(\phi) \subseteq F$. Then $\tau_{M,P}(\tau_{M,P}^{i}(\phi)) \subseteq \tau_{M,P}(F) = F$, by monotonicity and the assumption on F. □

The next step is the introduction of the important notion of *continuity*. This concept, which plays an important role in the more advanced parts of the theory of mathematical semantics, is encountered here as a property of the functionals σ_M or $\sigma_{M,P}$, as defined in definitions 3.6 and 3.8.

DEFINITION 3.13. Let V be any domain, and let σ be a monotonic functional over V (i.e., $\sigma \in (V \to V) \to (V \to V)$, and $\sigma(F) \subseteq \sigma(G)$ whenever $F \subseteq G$). σ is called *continuous* if, for each chain $\{F_i\}_{i=0,1,\ldots}$, with lub $\cup_{i=0}^{\infty} F_i$, we have that

$$\sigma(\overset{\infty}{\underset{i=0}{\cup}} F_i) = \overset{\infty}{\underset{i=0}{\cup}} \sigma(F_i).$$

REMARK. The lub in the right-hand side is well-defined, since, by monotonicity of σ, $\sigma(F_i) \subseteq \sigma(F_{i+1})$, $i=0,1,\ldots$; hence, $\{\sigma(F_i)\}_{i=0,1,\ldots}$ is indeed a chain.

THEOREM 3.14. *Let* S *be a program scheme,* M *an interpretation, and* σ_M *or* $\sigma_{M,P}$ *as in definitions* 3.6 *and* 3.8. *Then* σ_M *and* $\sigma_{M,P}$ *are continuous functionals.*

PROOF. The proof, which proceeds by formula induction on S is omitted. (It is presented e.g. in [1].) □

We are now sufficiently prepared for the important proof rule, due to SCOTT [12], which will be our main tool in the next section. For numerous other applications of the rule see e.g. [1,3,4,8,9]. The formulation given here is slightly more general than the one of [1,3,4].

One more notational convention is needed. Let $\Phi = \{S_1 \subseteq S_2, T_1 = T_2, \ldots\}$ be a formula as defined in definition 3.4. Then $\Phi[T/X] =$ $\{S_1[T/X] \subseteq S_2[T/X],\ T_1[T/X] = T_2[T/X], \ldots\}$.

THEOREM 3.15 (Scott's induction rule).
Let $M = \langle V, C, E, D\rangle$ *be an interpretation,* $\langle P,T\rangle \in D$, *and let* Φ *be any formula. Then, if*

(3.2) $\models_M \Phi[\Omega/X]$

and, for all F,

(3.3) $\Phi \models_{M\{F/X\}} \Phi[T[P\to X]/X]$

then

(3.4) $\models_M \Phi[P/X]$.

PROOF. Clearly, it is sufficient to show the assertion for the case that Φ consists of only one atomic formula, $T_1 \subseteq T_2$, say. Choose any M. We must show that from the assumptions it can be inferred that $M(T_1[P/X]) \subseteq M(T_2[P/X])$. By lemma 2.8, this is equivalent to showing that $M\{M(P)/X\}(T_1) \subseteq M\{M(P)/X\}(T_2)$. Let us put (cf. definitions 3.6 and 3.8) $\tau_1 = \tau_{1M,X}$, $\tau_2 = \tau_{2M,X}$, $\tau = \tau_{M,P,X}$. Using this notation, what we have to show reads: $\tau_1(M(P)) \subseteq \tau_2(M(P))$, or, by theorem 3.11, $\tau_1(\cup_{i=0}^{\infty}\tau^i(\phi)) \subseteq \tau_2(\cup_{i=0}^{\infty}\tau^i(\phi))$, or, by continuity, that $\cup_{i=0}^{\infty}\tau_1(\tau^i(\phi)) \subseteq \cup_{i=0}^{\infty}\tau_2(\tau^i(\phi))$. Thus, it is sufficient to show that $\tau_1(\tau^i(\phi)) \subseteq \tau_2(\tau^i(\phi))$, for each $i=0,1,\ldots$ This is done by induction on i.

a. $i = 0$. Proof of $\tau_1(\phi) \subseteq \tau_2(\phi)$.
   By assumption, $\models_M T_1[\Omega/X] \subseteq T_2[\Omega/X]$, i.e., $M(T_1[\Omega/X]) \subseteq M(T_2[\Omega/X])$, or $M\{M(\Omega)/X\}(T_1) \subseteq M\{M(\Omega)/X\}(T_2)$, or $M\{\phi/X\}(T_1) \subseteq M\{\phi/X\}(T_2)$, i.e., $\tau_1(\phi) \subseteq \tau_2(\phi)$.

b. Assume $\tau_1(\tau^i(\phi)) \subseteq \tau_2(\tau^i(\phi))$. Then, by definition of τ_1,τ_2:
   $M\{\tau^i(\phi)/X\}(T_1) \subseteq M\{\tau^i(\phi)/X\}(T_2)$. By (3.3), with $F = \tau^i(\phi)$, we infer
   $M\{\tau^i(\phi)/X\}(T_1[T[P\to X]/X]) \subseteq M\{\tau^i(\phi)/X\}(T_2[T[P\to X]/X])$, or, by lemma 2.8,
   $M\{\tau^i(\phi)/X\}\{M\{\tau^i(\phi)/X\}(T[P\to X])/X\}(T_1) \subseteq M\{\tau^i(\phi)/X\}\{M\{\tau^i(\phi)/X\}(T[P\to X])/X\}(T_2)$,
   or, by definition of τ, $M\{\tau^i(\phi)/X\}\{\tau(\tau^i(\phi))/X\}(T_1) \subseteq$
   $M\{\tau^i(\phi)/X\}\{\tau(\tau^i(\phi))/X\}(T_2)$, or, since $M\{G/X\}\{F/X\} = M\{F/X\}$,
   $M\{\tau^{i+1}(\phi)/X\}(T_1) \subseteq M\{\tau^{i+1}(\phi)/X\}(T_2)$, i.e., $\tau_1(\tau^{i+1}(\phi)) \subseteq \tau_2(\tau^{i+1}(\phi))$,
   which settles the induction step.

Hence, the justification of Scott's rule is complete. □

REMARK. Our present formulation of Scott's rule is more general than the one of [1,3,4], since it is of the form $\forall M[A(M) \wedge B(M) \to C(M)]$, whereas the one used before was of the form $\forall M[A(M)] \wedge \forall M[B(M)] \to \forall M[C(M)]$.

4. E.W. DIJKSTRA ON SEMANTICS

We apply the theory developed in sections 2 and 3 to provide a framework in which we hope to explain Dijkstra's approach to semantics. As starting point, he takes Hoare's approach, which deals with program correctness formulated in terms of expressions of the form {p} S {q}, which are to be read as: For all states x and y, if x satisfies property p, and program S changes x to y, then y satisfies q. In our formalism, we would like to state this as follows: Let S be a program scheme, p,q two boolean symbols, and M an interpretation. Then the correctness statement becomes:

(4.1) $\models_M \forall x,y[p(x) \wedge xSy \rightarrow q(y)]$.

There is a slight technical problem here, in that we have not formally defined the $\ldots\models_M\ldots$ notation for this case. (Remember that, sofar, we used this only with $\Phi \models_M \Psi$, with Φ,Ψ formulas, as in definition 3.4.) However, it is straightforward to extend our definition to deal with such mixture of the program scheme language and the language of predicate calculus, and we shall assume this done from now on. With (4.1) we have formulated what is usually called *partial correctness* of S with respect to p and q (in M). In Dijkstra's approach, it is not so much partial, but *total* correctness which is used: S is totally correct with respect to p and q (in M), if

(4.2) $\models_M \forall x[p(x) \rightarrow \exists y[xSy \wedge q(y)]]$,

i.e., total correctness implies, besides partial correctness, also *termination*. In order to explain the way in which the notion of total correctness is used in [5], we give the following quotation: "We consider the semantics of a program S fully determined when we can derive for any postcondition q satisfied by the final state, the weakest precondition that for this purpose should be satisfied by the initial state. We regard this weakest precondition as a function of the postcondition q, and denote it by "fS(q)". Here we regard the fS as a "predicate transformer", as a rule for deriving the weakest precondition from the postcondition to which it corresponds".

We grant the suspicious reader that it is not stated explicitly here that the precondition should guarantee termination. However, we do not see how to explain the properties of fS mentioned below (in particular D_2) if termination were not implied.

Now what is the weakest precondition for given S and q? From (4.2) we see that for *each* precondition p which guarantees total correctness, we have that $p(x) \rightarrow \exists y[xSy \wedge q(y)]$. Hence, whatever precondition p we choose, we always have that, if p(x), then $\exists y[xSy \wedge q(y)]$ holds. This suggests to us that the weakest precondition, in each x, is nothing but $\exists y[xSy \wedge q(y)]$. This explanation finds support in

- All postulates and rules of [5] become provable.
- A modified version of the Fundamental Theorem of [5] is also provable.

First we consider the four "basic properties" which are attributed to fS in [5].

D_1: "p = q implies fS(p) = fS(q)".

In our notation, this is nothing but the obvious implication:

For each M

$\models_M \forall x[p(x) \leftrightarrow q(x)] \rightarrow \forall x[\exists y[xSy \wedge p(y) \leftrightarrow \exists y[xSy \wedge q(y)]]$.

D_2: Let f,t be two special predicates which are false and true, respectively, in all possible states, i.e., let, for each $M = \langle V,C,E,D\rangle$, $M(f)(x) = \text{false}$, $M(t)(x) = \text{true}$, for each $x \in V$. D_2 now reads

"fS(f) = f",

or, in our notation,

For all M,

$\models_M \forall x[\exists y[xSy \wedge f(y)] \leftrightarrow f(x)]$,

which is easily seen to be true.

D_3: "fS(p∧q) = fS(p) ∧ fS(q)", or,

For all M,

$\models_M \forall x[\exists y[xSy \wedge p(y) \wedge q(y)] \leftrightarrow \exists y[xSy \wedge p(y)] \wedge \exists y[xSy \wedge q(y)]]$.

Since we have restricted our M to yield *functions* only, we easily establish D_3. (Note that for 'non-deterministic" programs, with $M(S)$ a relation instead of a function, D_3 fails.)

D_4: "$fS(p \vee q) = fS(p) \vee fS(q)$".
Left to the reader.

After these basic properties, several additional properties and rules are postulated or derived in [5]. We list only a selection:

D_5: "$p \Rightarrow q$ implies $fS(p) \Rightarrow fS(q)$".
This is again very clear. (Not clear is, to us, why D_5 is considered less basic than D_1.)

D_6: "$f(S_1;S_2)(p) = fS_1(fS_2(p))$", or
For all M,
$\models_M \forall x[\exists y[xS_1;S_2y \wedge p(y)] \leftrightarrow \exists y[xS_1y \wedge \exists z[yS_2z \wedge p(z)]]]$.
This is established as follows:
$\exists y[xS_1y \wedge \exists z[yS_2z \wedge p(z)]] \leftrightarrow$
$\exists y,z[xS_1y \wedge yS_2z \wedge p(z)] \leftrightarrow$
$\exists z[\exists y[xS_1y \wedge yS_2z] \wedge p(z)] \leftrightarrow$ (lemma 2.6e)
$\exists z[xS_1;S_2z \wedge p(z)]$.

D_7: "$f(\underline{\text{if}}\ p\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2)(q) = p \wedge fS_1(q) \vee \neg p \wedge fS_2(q)$".
This is also left to the reader.
Hint: Note that, for all M,
$\models_M \forall x,y[x\ \underline{\text{if}}\ p\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ y \leftrightarrow p(x) \wedge xS_1y \vee \neg p(x) \wedge xS_2y]$.

We now turn to the main theorem of [5], the so-called Fundamental Invariance Theorem for Recursive Procedures. We quote from [5]: "Consider a text, called H", of the form H": ...H'...H'...H'..., to which corresponds a predicate transformer fH", such that for a specific pair of predicates q and r, the assumption $q \Rightarrow fH'(r)$ is a sufficient assumption about the healthy fH' for proving $q \Rightarrow fH''(r)$. In that case, the recursive procedure H defined by <u>proc</u> H: ...H...H...H... <u>corp</u> (where we get this text by removing the dashes and enclosing the resulting text between the brackets <u>proc</u> and <u>corp</u>) enjoys the property that $q \wedge fH(t) \Rightarrow fH(r)$".

The first step in our analysis of this theorem is to rewrite, for some given M, $q \wedge fH(t) \Rightarrow fH(r)$, as follows:
$\models_M \forall x[q(x) \wedge fH(t)(x) \rightarrow fH(r)(x)]$, or
$\models_M \forall x[q(x) \wedge \exists y[xHy \wedge t(y)] \rightarrow \exists y[xHy \wedge r(y)]]$.

By the definition of t, and the restriction to *functions*, this is equivalent to

(4.3) $\models_{M} \forall x,y[q(x) \wedge xHy \rightarrow r(y)]$.

Therefore, we can rewrite the theorem in our notation as: Let H, H', H" be as above, let $T \equiv \ldots H \ldots H \ldots H \ldots$, and let $M = \langle V, C, E, D \rangle$ be any interpretation with $\langle H,T \rangle \in D$. (We assume that $H \in P$.)
If

(4.4) $\forall x[q(x) \rightarrow fH'(r)(x)] \models_{M} \forall x[q(x) \rightarrow fH''(r)(x)]$

then

(4.5) $\models_{M} \forall x,y[q(x) \wedge xHy \rightarrow r(y)]$.

Choosing for M an interpretation such that, for all x, $M(q)(x)$ = true and $M(r)(x)$ = false, we easily see (using D_2) that such M satisfies (4.4). In fact, (4.4) reduces in this case to: If, for all $x \in V$, true implies false, then, for all $x \in V$, true implies false. This is clearly satisfied, and we may therefore infer that (4.5) for this M also holds, i.e., that, for all $x,y \in V$, (true and $xM(H)y$) implies false. This is equivalent to asserting that, for no $x,y \in V$, $xM(H)y$ holds, and this is a contradiction: It amounts to asserting that, for each procedure H, H does not terminate for any input state (whatever choice we make for its body T).

The final stage in our analysis is the derivation of the corrected version of the theorem. We first state the new version in Dijkstra's notation: "If $q \wedge fH'(t) \Rightarrow fH'(r)$ is a sufficient assumption to prove $q \wedge fH''(t) \Rightarrow fH''(r)$, then $q \wedge fH(t) \Rightarrow fH(r)$ holds". We shall provide this statement with an explanation which leads to a true assertion in our framework. The first (not yet satisfactory) attempt to translate this in our notation, using the reduction (4.3), yields: For all M, if

$$\forall x,y[q(x) \wedge xH'y \rightarrow r(y)] \models_{M} \forall x,y[q(x) \wedge xH''y \rightarrow r(y)]$$

then

$$\models_{M} \forall x,y[q(x) \wedge xHy \rightarrow r(y)].$$

In order to make this more manageable, we start with the introduction of a new notation, which allows us to rewrite the above in a simpler form. Let, for each boolean symbol p, $\tilde{p}$ be an elementary action symbol, with the convention that, for all M, $M(\tilde{p})(x) = x$, whenever $M(p)(x) = \text{true}$, and $M(\tilde{p})(x)$ is undefined, for all other x. We then obtain as next version of the theorem: For all M, if

$$\tilde{q};H' \subseteq H';\tilde{r} \models_M \tilde{q};H'' \subseteq H'';\tilde{r}$$

then

$$\models_M \tilde{q};H \subseteq H;\tilde{r}.$$

This is not yet what we want, since we still have to clarify the role of H' and H". Remember that H" was the name of the piece of program text ...H'...H'...H'... . The quotation from [5] above does not specify what H' stands for. It seems natural, however, to see it as a program scheme variable, which has to be "quantified" in the sense made more precise in our next stage: For all M, if, for all F,

$$\tilde{q};H' \subseteq H';\tilde{r} \models_{M\{F/H'\}} \tilde{q};H'' \subseteq H'';\tilde{r}$$

then

$$\models_M \tilde{q};H \subseteq H;\tilde{r}.$$

This is the decisive step, which achieves what we want. Summarizing, we put
- H into the class of procedure symbols,
- H' into the class of program scheme variables,
- $T \equiv$ ...H...H...H..., and, hence,
- $T[H \rightarrow H'] \equiv$ ...H'...H'...H'..., i.e., $T[H \rightarrow H'] \equiv H''$,

and obtain: For all M, if, for all F,

$$\tilde{q};H' \subseteq H';\tilde{r} \models_{M\{F/H'\}} \tilde{q};H'[T[H\to H']/H'] \subseteq H'[T[H\to H']/H'];\tilde{r},$$

then

$$\models_M \tilde{q};H'[H/H'] \subseteq H'[H/H'];\tilde{r}.$$

Observing that, trivially,

$$\models_M \tilde{q};H'[\Omega/H'] \subseteq H'[\Omega/H'];\tilde{r},$$

we finally see that the theorem becomes: For all M, if

$$\models_M \tilde{q}\,;H'[\Omega/H'] \subseteq H'[\Omega/H'];\tilde{r},$$

and, for all F,

$$\tilde{q}\,;H' \subseteq H';\tilde{r} \models_{M\{F/H'\}} \tilde{q};H'[T[H\to H']/H'] \subseteq H'[T[H\to H']/H'];\tilde{r},$$

then

$$\models_M \tilde{q};H'[H/H'] \subseteq H'[H/H'];\tilde{r},$$

and this is nothing but an instance of Scott's induction rule, with $\Phi = \{\tilde{q};H' \subseteq H';\tilde{r}\}$. □

REFERENCES

[1] 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, 1975.

[2] DE BAKKER, J.W., *Least fixed points revisited,* to appear in Theoretical Computer Science.

[3] 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.

[4] 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.

[5] DIJKSTRA, E.W., *A simple axiomatic basis for programming language constructs*, Indagationes Mathematicae, 36 (1974) 1-15.

[6] HOARE, C.A.R., *An axiomatic basis for computer programming*, C.ACM, 12 (1969), 576-580.

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

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

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

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

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

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

[13] 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.