stichting
mathematisch
centrum

MC

DEPARTMENT OF COMPUTER SCIENCE IW 50/76 JUNE

J.W. DE BAKKER

TERMINATION OF NONDETERMINISTIC PROGRAMS

First printing October 1975
Second printing July 1976

2e boerhaavestraat 49 amsterdam

BIBLIOTHEEK MATHEMATISCH CENTRUM
——AMSTERDAM——

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

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.

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

ACM-Computing Reviews-categories: 5.24

CONTENTS

TERMINATION OF NONDETERMINISTIC PROGRAMS *)

by

J.W. de Bakker

ABSTRACT

A new formalism to deal with program termination in the presence of both nondeterminacy and recursion is presented. For the denotational (least-fixed-point) semantics of programs involving these concepts, we cannot use the customary set-theoretical ordering between the input-output relations associated with programs. A new ordering definition, due to Egli, is applied instead. Next, we describe our method of expressing termination of programs built up using sequential composition, nondeterministic choice, selection and recursion. The method is justified in the framework of denotational semantics. Finally, it is compared to the theory of Hitchcock & Park - which uses well-founded relations and program derivatives - and a new proof of an extended version of their main theorem is presented.

KEY WORDS & PHRASES: *Program termination, nondeterminacy, recursion, least-fixed-point semantics, well-founded relations, program derivatives.*

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

NOTATIONS

Section 2

$A = \{A_1, A_2, \ldots\}$ elements denoted by $A, A_i, \ldots$	program scheme constants
$B = \{b_1, b_2, \ldots\}$ elements denoted by $b, b_i, \ldots$	boolean scheme constants
$X = \{X_1, X_2, \ldots\}$ elements denoted by $X, X_i, \ldots$	program scheme variables
$Y = \{Y_1, Y_2, \ldots\}$ elements denoted by $Y, Y_i, \ldots$	boolean scheme variables
$\sigma \in S$, $\pi \in P$	program schemes, boolean schemes
$\sigma_1 \cup \sigma_2$, $\sigma_1;\sigma_2$, $\underline{\text{if}}\ b\ \underline{\text{then}}\ \sigma_1\ \underline{\text{else}}\ \sigma_2$, $\mu X[\sigma]$	program scheme construction rules
$\pi_1 \wedge \pi_2$, $\sigma \to \pi$, $\underline{\text{if}}\ b\ \underline{\text{then}}\ \pi_1\ \underline{\text{else}}\ \pi_2$, $\mu Y[\pi]$	boolean scheme construction rules
$\sigma_1 \equiv \sigma_2$, $\sigma_1 \not\equiv \sigma_2$, $\pi_1 \equiv \pi_2$, $\pi_1 \not\equiv \pi_2$ $\sigma[\tau/X]$, $\pi[\tau/X]$, $\pi[\pi_1/Y]$	syntactic (non)identity substitution

Section 3

$V = V_0 \cup \{\perp\}$	domain of states, with $\perp$ the undefined state
$W = \{\underline{\text{true}}, \underline{\text{false}}\}$	
$R, S, \ldots \quad \subseteq V \times V$	binary relations
$p, q, \ldots \quad : V \to W$	predicates
$R;S$, $R \cup S$, $\underline{\text{if}}\ p\ \underline{\text{then}}\ R\ \underline{\text{else}}\ S$, $p \wedge q$, $R \to p$, $\underline{\text{if}}\ p\ \underline{\text{then}}\ q\ \underline{\text{else}}\ r$	operations upon relations and predicates

$TR(V)$	total extended binary relations over V
$HE(V)$	extended predicates: $V \to W$
$M = \langle V,C,E \rangle$	interpretation
$M_O(\sigma)$	operational interpretation
$M(\sigma)$, $M(\pi)$	denotational interpretation

Section 4

$\leq$	ordering relation on $TR(V)$ and $HE(V)$
Φ, Ψ	operators: $TR(V) \to TR(V)$, $HE(V) \to HE(V)$
$\mu\Phi$, $\mu\Psi$	least fixed points of Φ, Ψ
$\bigwedge_{i \in I} p_i$	greatest lower bound
$M\{R/X\}$, $M\{p/Y\}$	variants of an interpretation
$\lambda R \cdot \ldots R \ldots$, $\lambda p \cdot \ldots p \ldots$	λ-notation
$\bigvee_{i=0}^{\infty} R_i$	least upper bound
O	least element of $TR(V)$

Section 5

$e(R)$	R terminates properly
$\tilde{\sigma}$	boolean termination scheme for σ

Section 6

Ω, Δ, ω, θ	special constants in A, B
O, I, f, t	special relations and predicates
$\sigma \cup \pi$, $\bar{\pi}$, $\sigma \circ \pi$, $\pi_1 \vee \pi_2$	extended construction rules and
$S \cup p$, $\bar{p}$, $S \circ p$, $p \vee q$	operators
$\mathring{\sigma}$, $\underset{\circ}{\sigma}$, $(\mathring{\sigma}, \underset{\circ}{\sigma})$	upper- and lower derivatives (taken "in $\mu X[\sigma]$")

$\iota(\sigma)$	well-founded part of a scheme
$\models_M \sigma = \tau$ ($\models \sigma = \tau$)	σ and τ are equivalent under M (all consistent M)

1. INTRODUCTION

In a paper presented by P. HITCHCOCK & D. PARK at the first Colloquium on Automata, Languages and Programming [7], a very interesting method was proposed to attack the problem of proving program termination. Two main innovations of this paper were: First, the introduction of well-founded relations as a tool; this necessitated consideration of non-continuous - though monotonic - operators, and was an important conceptual extension of hitherto proposed methods. Secondly, the paper introduced the notion of (upper- and lower-) derivative of a program (or, rather, a program scheme) and introduced a formalism to express termination in terms of these derivatives. However, this formalism was applied only to *deterministic* programs (i.e., programs determining *single-valued* state-transforming functions). Moreover, for a long time we have felt that it should be possible to replace the derivative-formalism with a more direct approach.

The present paper is an attempt at solving both problems: Firstly, to develop a formalism which can deal with termination of *nondeterministic* programs as well. Secondly, to avoid the notion of derivative and to use a more intuitively appealing technique.

The first problem could be solved only after we became aware of a paper of EGLI [6] which turned out to be essential for our investigation. We here sketch in which way Egli's idea is applied: Programs σ determine state transformations S, and the presence of nondeterminacy implies that it is necessary to use binary *relations* to describe these transformations: For states x, y_1, y_2 one may have xSy_1 and xSy_2 with $y_1 \neq y_2$. However, the simple presence of some y such that xSy holds, does not guarantee termination of *all* computations determined by σ. It may well be that one path of the computation delivers a value, whereas some other path leads to a nonending computation. So we add - as is often done in this type of considerations - one *new* state - denoted by $\perp$, say - which stands for "undefined", and we define our computations such that if σ determines some nonterminating computation, then $x S \perp$ holds, besides, possibly, $xSy_1, xSy_2, \ldots$. However, this renders it impossible to use set-theoretic inclusion between relations as a model of approximation between programs. If σ_1, σ_2 determine relations S_1, S_2 such that $S_1 \subseteq S_2$, then there is *no* reason to view σ_2 as providing

more information than σ_1: Observe that, e.g., $\{\langle x,y\rangle\} \subseteq \{\langle x,y\rangle\ \langle x,\perp\rangle\}$. This has the following undesirable effect: Adding the possibility of an undefined computation for input x increases the information about the program. Hence, the ordering "$\subseteq$" is not appropriate to capture the intuition we want to model. On the other hand, some ordering *is* needed in order that the usual techniques of denotational semantics - interpreting recursive programs as *least* fixed points of certain operators; note that this implies some ordering - be applicable, and it is precisely at this point that Egli's idea gives the desired solution. His ordering definition - discussed in section 4 - enables us to give a denotational semantics of programs involving both nondeterminacy and recursion.

This having been satisfactorily settled, we can then turn to our main problem: To develop a formalism describing program termination which is intuitively appealing and which also allows us to clarify some aspects of the Hitchcock & Park theory. We present our method in the framework of program schemes - with sequential composition, nondeterministic choice, selection and recursion as construction rules - and we define a way of describing, with each program scheme, a (boolean) scheme expressing its termination properties. It seems to us that the simplicity of this definition is a main advantage of our theory. E.g., a certain new operation introduced in order to deal with termination of schemes constructed through sequential composition, leads quite naturally to an explanation of the role of well-foundedness of relations in the theory of [7]. Finally, the main theorem of [7] can be rather nicely derived on the base of our method, yielding an extended version which also covers the nondeterministic case.

The paper is organized as follows:
Section 2 gives the syntax of program schemes and of boolean schemes. Moreover, the concept of substitution is introduced.
Section 3 deals with the semantics of schemes. Two ways of interpreting a program scheme are provided. First, the *operational* definition which uses the notion of *computation sequence*, and which stays close to the customary way of explaining the meaning of the programming concepts involved. Next, we turn to the *denotational* semantics of both program schemes and boolean schemes. This becomes interestingly different from operational semantics only in the case of recursion, dealt with in
Section 4. The more or less standard material on the least-fixed-point

interpretation of recursion is presented. However, a rather careful development of this is needed since first of all a new ordering is involved - the above mentioned one of Egli - and, secondly, because in our definition of *boolean* schemes we have introduced a non-continuous - though monotonic - operation.

Section 5 is the central one of our paper. It presents our definition of a process of associating, with each program scheme, a boolean scheme expressing its termination properties. Moreover, the validity of the definition is proved using the techniques of section 4.

Section 6 finally contains a description of the Hitchcock & Park formalism and our method of proving their main theorem.

The present paper is specifically devoted to theoretical considerations, aiming at an understanding of the interrelationship between the three important concepts of recursion, nondeterminacy and termination, rather than at the introduction of new practical techniques for program proving.

As related work dealing with formalisms for program termination we mention: Manna's notion of total correctness, described e.g. in [8] (see also a comment on this we make at the end of [5]). Furthermore, there is the axiomatic presentation given by MANNA & PNUELI (see e.g. [8]) of the classical idea of using well-founded sets (Turing, Floyd). The connection between the Manna & Pnueli proof rule and the Hitchcock & Park theory was clarified in our [3]. Altogether, only a small number of formalisms have been proposed sofar, and we hope that the present paper will stimulate further work in this interesting and difficult problem area.

2. SYNTAX

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, more specifically to study program termination. A program scheme is a linguistic object - i.e., a sequence of symbols structured in a certain way - which serves as an abstract version of an ordinary program. This should be taken in the sense that in a scheme one abstracts from an analysis of the elemen-

tary statements which make up the program: There is a class of elementary actions - *program scheme constants* as they will be called - which in our system are considered atomic whereas in a real-life program they would be further specified as, e.g., assignment statements.

For reasons which are more of technical than of fundamental nature - they stem from our way of incorporating recursion in the system - we also need to have available a class of *program scheme variables*.

Furthermore, we introduce, besides the class of program schemes, also the class of *boolean schemes*. Whereas program schemes are to be interpreted (section 3) as state-transforming functions - or, rather, as binary relations, because of nondeterminacy - boolean schemes are interpreted as functions from the set of states to the set of truth values, {true,false}, say. It should be emphasized that in our approach boolean schemes are introduced only as a tool to investigate termination properties of program schemes. In particular, they enable us to make certain formal statements about these properties. The boolean schemes are *not* themselves to be seen as abstractions of ordinary programs: their definition includes a non-continuous (section 4) - and, therefore, by "Scott's thesis" [10], non-computable - operation.

We first give the notation for constants and variables:

NOTATION 2.1 (Constants and variables).

a. The set $A = \{A_1, A_2, \ldots\}$ is the set of *program scheme constants*. Arbitrary elements of A are denoted by $A, A_i, \ldots$.

b. The set $B = \{b_1, b_2, \ldots\}$ is the set of *boolean scheme constants*. Arbitrary elements of B are denoted by $b, b_i, \ldots$.

c. The set $X = \{X_1, X_2, \ldots\}$ is the set of *program scheme variables*. Arbitrary elements of X are denoted by $X, X_i, \ldots$.

d. The set $Y = \{Y_1, Y_2, \ldots\}$ is the set of *boolean scheme variables*. Arbitrary elements of Y are denoted by $Y, Y_i, \ldots$.

From these classes of symbols, program schemes and boolean schemes are made up according to certain construction rules given in

DEFINITION 2.2 (Schemes). The class of program schemes S and the class of boolean schemes P are defined as follows:

a. Each program scheme constant and each program scheme variable is an element of S.

b. Each boolean scheme constant and each boolean scheme variable is an element of P.

c. If $\sigma,\sigma_1,\sigma_2 \in S$, $b \in B$, $X \in X$, then

$(\sigma_1;\sigma_2)$	: *sequential composition*
$(\sigma_1 \cup \sigma_2)$	: *nondeterministic choice*
$\underline{\text{if}}\ b\ \underline{\text{then}}\ \sigma_1\ \underline{\text{else}}\ \sigma_2\ \underline{\text{fi}}$	: *selection*
$\mu X[\sigma]$	: *recursion*

are elements of S.

d. If $\pi,\pi_1,\pi_2 \in P$, $\sigma \in S$, $b \in B$, $Y \in Y$, then

$(\pi_1 \wedge \pi_2)$	: *conjunction*
$(\sigma \to \pi)$	: to be explained later
$\underline{\text{if}}\ b\ \underline{\text{then}}\ \pi_1\ \underline{\text{else}}\ \pi_2\ \underline{\text{fi}}$	: *selection*
$\mu Y[\pi]$	: *recursion*

are elements of P.

Examples.

c. $\underline{\text{if}}\ b\ \underline{\text{then}}\ (A_1;X)\ \underline{\text{else}}\ A_2\ \underline{\text{fi}}$, $\mu X[X]$, $\mu X[(((A_1;X);A_2) \cup A_3)]$.

d. $(b_1 \wedge b_2)$, $\underline{\text{if}}\ b_1\ \text{then}\ (A \to b_2)\ \underline{\text{else}}\ Y\ \underline{\text{fi}}$, $\mu Y[A \to Y)]$.

By way of explanation of this definition we remark that

1. Sequential composition, conjunction and selection define the usual operations which need no further comment.
2. Nondeterministic choice is a central operation in the system: $(\sigma_1 \cup \sigma_2)$ specifies that either σ_1 or σ_2 is to be performed (not both!), but which of the two is left open.
3. The reader who is not accustomed to the μ-notation for recursion (see [1,2,7]) may be helped by the following explanation: Consider the scheme $\mu X[\sigma]$. Here σ is any scheme which may have occurrences of the variable X, i.e., $\sigma \equiv \sigma[X]$, writing informally. Now the intended meaning of $\mu X[\sigma]$ is the same as that - in a more customary notation - of a call of the recursive procedure P declared by $\underline{\text{proc}}\ P;\ \sigma[P]$, where $\sigma[P]$ results from $\sigma[X]$ by replacing all occurrences of X by P. For example, execution of $\mu X[\underline{\text{if}}\ b\ \underline{\text{then}}\ (A_1;X)\ \underline{\text{else}}\ A_2\ \underline{\text{fi}}]$ amounts to a call of the procedure P declared by $\underline{\text{proc}}\ P;\ \underline{\text{if}}\ b\ \underline{\text{then}}\ (A_1;P)\ \underline{\text{else}}\ A_2\ \underline{\text{fi}}$.

4. Anticipating the definitions to be given below, where schemes σ are interpreted as binary relations S, and schemes π as predicates p, we already mention that the predicate $(S \to p)$ - as interpretation of $(\sigma \to \pi)$ - will obtain the following meaning: $(S \to p)(x)$ iff $\forall y[xSy \to p(y)]$.

Before we can give our next definition which introduces *substitution*, we need two more notations:

NOTATION 2.3 (Syntactical identity).
$\sigma_1 \equiv \sigma_2$ ($\sigma_1 \not\equiv \sigma_2$) denotes that σ_1 and σ_2 are identical (not identical) sequences of symbols.
$\pi_1 \equiv \pi_2$ and $\pi_1 \not\equiv \pi_2$ are defined similarly.

NOTATION 2.4 (Bound and free occurrences).
All occurrences of a program scheme variable X in a scheme $\mu X[\sigma]$ are *bound*. An occurrence of a variable X_1 in a scheme σ_1 is called *free* iff it is not a bound occurrence. A scheme $\mu X_1[\sigma_1]$ which results from a scheme $\mu X[\sigma]$ by replacing all occurrences of X in the latter by some X_1 which does not occur free in σ, is called a *rewritten* version of $\mu X[\sigma]$. Two schemes such that one is a rewritten version of the other will always be identified in the sequel. Mutatis mutandis these definitions also hold for boolean schemes.

DEFINITION 2.5 (Substitution). For $\sigma,\tau \in S$, $X \in X$, we define the operation $\sigma[\tau/X]$: τ is substituted for X in σ, by induction on the structure of σ:

a. $\sigma \equiv X$: $\sigma[\tau/X] \equiv \tau$
b. $\sigma \in A \cup X$, $\sigma \not\equiv X$: $\sigma[\tau/X] \equiv \sigma$
c. $\sigma \equiv (\sigma_1;\sigma_2)$: $\sigma[\tau/X] \equiv (\sigma_1[\tau/X];\sigma_2[\tau/X])$
d. $\sigma \equiv (\sigma_1 \cup \sigma_2)$: $\sigma[\tau/X] \equiv (\sigma_1[\tau/X] \cup \sigma_2[\tau/X])$
e. $\sigma \equiv$ **if** b **then** σ_1 **else** σ_2 **fi**: $\sigma[\tau/X] \equiv$ **if** b **then** $\sigma_1[\tau/X]$ **else** $\sigma_2[\tau/X]$ **fi**.
f. $\sigma \equiv \mu X_1[\sigma_1]$: $\sigma[\tau/X] \equiv \mu X_1[\sigma_1[\tau/X]]$
provided that $X \not\equiv X_1$, and that X_1 does not occur free in τ; these conditions can always be made to be satisfied by suitably rewriting the bound variable X_1 in $\mu X_1[\sigma_1]$.

Now let $\pi,\pi_1 \in P$ and $Y \in Y$. The definitions of $\pi[\pi_1/Y]$ and $\pi[\tau/X]$ are straightforward variations on that of $\sigma[\tau/X]$ and therefore omitted. Note that a boolean scheme may have free occurrences of a program scheme variable, due to the construct $(\sigma \to \pi)$.

3. SEMANTICS

3.1. *Introduction*

Schemes are provided with meaning by a process of *interpretation*, which maps schemes $\sigma \in S$ to binary relations S, and schemes $\pi \in P$ to predicates (unary relations) p.

First we introduce the notation for relations and predicates, and for some operations upon these we find useful.

NOTATION 3.1 (Relations and predicates).

1. Let V be any nonempty set - in the sequel always called the set of *states* - with elements x,y,..., and let W be the set of truth values: W = {true,false}. Binary relations (over V), denoted by R,S,..., are subsets of $V \times V$, and predicates (over V), denoted by p,q,..., are *total* functions from V to W.
2. For R,S,... binary relations, p,q,r,... predicates, x,y,z $\in V$, we define
   a. $R;S = \{\langle x,y\rangle \mid \exists z[xRz \wedge zRy]\}$
      $R \cup S = \{\langle x,y\rangle \mid xRy \vee xSy\}$
      if p then R else S fi $= \{\langle x,y\rangle \mid p(x) \wedge xRy \vee \neg p(x) \wedge xSy\}$.
   b. $(p \wedge q)(x)$ iff $p(x) \wedge q(x)$
      $(R \rightarrow p)(x)$ iff $\forall y[xRy \rightarrow p(y)]$
      if p then q else r fi iff $p(x) \wedge q(x) \vee \neg p(x) \wedge r(x)$.

Our definition of the interpretation of a scheme is organized as follows: First we give an intuitive explanation of the issues involved in the definition. Then we give (section 3.2) a definition of *operationally* interpreting schemes $\sigma \in S$ through the introduction of the notion of *computation sequence*. The operational definition is intended to embody the meaning of the various programming concepts in a manner which is as close as possible to the way the programmer understands them. Thus, it serves as a transition to the more abstract definition which follows in section 3.3, and which will remain our main tool in the rest of the paper. This *denotational* interpretation avoids the use of computation sequences, and is justified by some of the results in section 3.2. The difference between the two approaches is in particular noticeable for recursive schemes. Due

to the need for more elaborate preparations, a separate section is devoted to these (section 4). Section 3.3 brings also the denotational interpretation rules for boolean schemes. (Remember that boolean schemes - which in our paper serve only as a tool to investigate program schemes - cannot be given an operational definition of their own.)

A first attempt at interpreting schemes σ might be to use state-transforming functions as intended meaning. However, because of the presence of nondeterminacy, we need multi-valued functions, so the relational formalism is the appropriate one. As we shall see, each scheme σ determines, in general, a number of possible computations, and, for S the interpretation of σ, and x any input state, we may have xSy for zero, one or more output states y. Because of our special interest in termination, we want to incorporate in the system one special state, for which we use the element denoted by "$\perp$", with the convention that $x S \perp$ holds iff some computation sequence specified by σ does not terminate properly. By this we mean that either the computation sequence is infinite, or that one of the elementary actions (interpreted elements of A or X) is undefined at some intermediate state. Thus, in the most general case we may have that both $xS\perp$, and $xSy_1, xSy_2, \ldots$ hold for given x, this meaning that there is (at least) one computation sequence specified by σ which does not terminate properly, and a number of other ones terminating with outputs $y_1, y_2, \ldots$.

In fact, the device used here is rather well-known in systems dealing with partial functions. Adding "$\perp$" as outcome is there also used to turn partial functions into total ones, which often is advantegeous. See [11] for further information.

However, in the relational approach there is one serious difficulty: Anticipating some of the considerations presented below to deal with recursion, we already mention that for the denotational treatment of this we need a partial ordering of the relations, written say as $R \leq S$, such that, in an intuitive sense, $R \leq S$ holds if (the program with interpretation) R *approximates* (the program with interpretation) S, i.e., iff S provides more information on the computation than R. However, in the approach using binary relations over the extended domain $V \cup \{\perp\}$, it is not possible to take for "$\leq$" the usual set-theoretic inclusion "$\subseteq$". Note that, e.g., $\{\langle x,y\rangle\} \subseteq \{\langle x,y\rangle, \langle x,\perp\rangle\}$, i.e., using this ordering a program

would provide more information when the possibility of a nonterminating computation were added, and that is certainly not in accordance with the intuition we want to capture. Therefore, we need a different definition of "$\leq$". This we found in a recent paper by EGLI [6], and it will be dealt with in detail in section 4.

We close this introductory section with a few more notations, the first of which slightly changes the notation for the set of states.

NOTATION 3.2 (Set of states). The set of states V is given as $V = V_0 \cup \{\perp\}$, where $\perp \notin V_0$, and V_0 is the set of proper states.

NOTATION 3.3 (Extended relations). For V as above, $TR(V)$ is the set of all *total, extended* binary relations over V, i.e., it consists of all binary relations over V such that both a and b hold:
a. $\forall x \in V \quad \exists y \in V[xRy]$
b. $\forall x \in V[\perp Rx \rightarrow x=\perp]$
(i.e., R is everywhere defined on V, and, for input $\perp$, $\perp$ is the only possible output).

NOTATION 3.4 (Extended predicates). For V as above, $HE(V)$ is the set of all (total,) extended predicates over V, i.e., it consists of all functions from V to W such that both a and b hold:
a. $\forall x \in V_0 \quad \exists y \in W[p(x)=y]$
b. $p(\perp) =$ <u>false</u>
(the motivation for clause b is discussed in section 5).

The introduction of V as $V = V_0 \cup \{\perp\}$ necessitates a slight adaptation of the definition of "$\rightarrow$" (the reasons for which will become clear later in the paper).

NOTATION 3.5 (Adapted definition of "$\rightarrow$"). Let $R \in TR(V)$, $p \in HE(V)$. We put
- for $x \in V_0$: $(R \rightarrow p)(x)$ iff $\forall y \in V_0[xRy \rightarrow p(y)]$
- for $x = \perp$: $(R \rightarrow p)(\perp) =$ <u>false</u>.

3.2. *Operational semantics*

In our definition of operational semantics we use the notion of compu-

tation sequence in a way which is very similar to that of some of our previous papers ([2,4,5], see also [9]). In fact, the definition given presently is a straightforward extension of its predecessors, and included here only for completeness sake. The reader who is already familiar with the kind of considerations we are concerned with here, may well skip the present section and immediately go on with the definition of denotational semantics (section 3.3) which is the only one to be used in the remainder of the paper.

An interpretation M is given as a triple $M = \langle V,C,E\rangle$, where
- $V = V_0 \cup \{\perp\}$ is as above.
- C (for *constants*) is a mapping from elements $A \in \mathcal{A}$ to binary relations over V_0, and from elements $b \in \mathcal{B}$ to predicates over V_0.
- E (for *environment* is a mapping from elements $X \in \mathcal{X}$ to binary relations over V_0, and from elements $Y \in \mathcal{Y}$ to predicates over V_0.

(Note that the definitions of C and E are with respect to V_0 only, i.e., the undefined element $\perp$ plays no role.)

Our task is now to specify, for given V,C,E, how to define M to yield, for each $\sigma \in S$, a total binary relation S in $TR(V)$. For this we need the notion of computation sequence:

DEFINITION 3.6 (Computation sequence). A computation sequence with respect to $M = \langle V,C,E\rangle$ is a construct of one of the following three forms:

(3.1) $\quad x_0\sigma_0x_1\sigma_1 \cdots x_{n-1}\sigma_{n-1}x_n$

with $x_i \in V_0$, $i = 0,1,\ldots,n$, and $\sigma_i \in S$, $i = 0,1,\ldots,n-1$
((3.1) is a sequence which *properly terminates*), or

(3.2) $\quad x_0\sigma_0x_1\sigma_1 \cdots x_m\sigma_m$

with $x_i \in V_0$, $i = 0,1,\ldots,m$, and $\sigma_i \in S$, $i = 0,1,\ldots,m$
((3.2) is a sequence which *improperly terminates*), or

(3.3) $\quad x_0\sigma_0x_1\sigma_1 \cdots x_p\sigma_p \cdots$

with $x_i \in V_0$, $i = 0,1,\ldots,p,\ldots$, and $\sigma_i \in S$, $i = 0,1,\ldots,p,\ldots$ ((3.3) is a sequence which is *nonterminating*), such that the following requirements are satisfied: For each 3-tuple $x_{n-1}\sigma_{n-1}x_n$, occurring at the end of a sequence (3.1), we have that either

- $\sigma_{n-1} \equiv A$, for some $A \in \mathcal{A}$, and $x_{n-1}C(A)x_n$, or
- $\sigma_{n-1} \equiv X$, for some $X \in \mathcal{X}$, and $x_{n-1}E(X)x_n$.

For each pair $x_m\sigma_m$, occurring at the end of a sequence (3.2), we have that either

- $\sigma_m \equiv A$, for some $A \in \mathcal{A}$, and there is no $y \in V_0$ such that $x_mC(A)y$, or
- $\sigma_m \equiv X$, for some $X \in \mathcal{X}$, and there is no $y \in V_0$ such that $x_mE(X)y$.

For each 4-tuple $x_i\sigma_ix_{i+1}\sigma_{i+1}$, occurring in (3.1), (3.2) or (3.3), we have that (exactly) one of the following conditions is satisfied:

a. $\sigma_i \equiv (\sigma' \cup \sigma'')$, $x_{i+1} = x_i$, and $\sigma_{i+1} \equiv \sigma'$ or $\sigma_{i+1} = \sigma''$.
   (This clause allows a nondeterministic choice between two ways of continuing the computation.)

b. $\sigma_i \equiv$ <u>if</u> b <u>then</u> σ' <u>else</u> σ'' <u>fi</u>, $x_{i+1} = x_i$, and $\sigma_{i+1} \equiv \sigma'$ if $C(b)(x) =$ <u>true</u>, $\sigma_{i+1} \equiv \sigma''$ if $C(b)(x) =$ <u>false</u>.

c. $\sigma_i \equiv \mu X[\sigma]$, $x_{i+1} = x_i$, and $\sigma_{i+1} \equiv \sigma[\mu X[\sigma]/X]$.
   (This clause is the copy-rule for procedure execution: Compare the case of the procedure declared by <u>proc</u> P;σ[P], where a call of P leads to execution of σ[P].)

d. $\sigma_i \equiv (A;\sigma)$, $x_iC(A)x_{i+1}$, and $\sigma_{i+1} \equiv \sigma$.

e. $\sigma_i \equiv (X;\sigma)$, $x_iE(X)x_{i+1}$, and $\sigma_{i+1} \equiv \sigma$.

f. $\sigma_i \equiv ((\sigma';\sigma'');\sigma)$, $x_{i+1} = x_i$, and $\sigma_{i+1} \equiv (\sigma';(\sigma'';\sigma))$.

g. $\sigma_i \equiv ((\sigma' \cup \sigma'');\sigma)$, $x_{i+1} = x_i$, and $\sigma_{i+1} \equiv ((\sigma';\sigma) \cup (\sigma'';\sigma))$.

h. $\sigma_i \equiv$ (<u>if</u> b <u>then</u> σ' <u>else</u> σ'' <u>fi</u>;σ), $x_{i+1} = x_i$, and $\sigma_{i+1} \equiv$ <u>if</u> b <u>then</u> $(\sigma';\sigma)$ <u>else</u> $(\sigma'';\sigma)$ <u>fi</u>.

i. $\sigma_i \equiv (\mu X[\sigma'];\sigma)$, $x_{i+1} = x_i$, and $\sigma_{i+1} \equiv (\sigma'[\mu X[\sigma']/X];\sigma)$.

By way of general explanation of the structure of the definition, we remark that at each moment during the computation, the scheme σ_i contains that part of the program scheme which is still to be executed with current state x_i. Computation may either terminate properly - with the execution of the final elementary action (interpreted element of $\mathcal{A}$ or $\mathcal{X}$), terminate improperly ($C(A)$ or $E(X)$ being undefined at the current state), or not

terminate at all. Once the reader has digested the formalism, we hope he will agree that all clauses of the definition are in accordance with his usual operational understanding of the programming concepts involved here.

We now define

DEFINITION 3.7 (Operational semantics). Let $\sigma \in S$, and let $M = \langle V,C,E\rangle$ be an interpretation. We define the operational meaning $M_O(\sigma)$ of the scheme σ as follows: For each $x,y \in V$ we put $xM_O(\sigma)y$ iff (at least) one of the following conditions is satisfied:

a. $x \in V_0$, $y \in V_0$, and there exists a computation sequence (3.1) with $x_0 = x$, $x_n = y$, and $\sigma_0 \equiv \sigma$.

b. $x \in V_0$, $y = \perp$, and there exists a computation sequence (3.2) or (3.3) with $x_0 = x$ and $\sigma_0 \equiv \sigma$.

c. $x = \perp$ and $y = \perp$.

That this definition has the desired properties can be seen from the following lemma's which we state without proof - which would proceed by a fairly straightforward induction on the structure of the schemes:

LEMMA 3.8. *For each M,*

a. $M_O(((\sigma_1;\sigma_2);\sigma_3)) = M_O((\sigma_1;(\sigma_2;\sigma_3)))$
(this associativity result allows us to omit parentheses).

b. $M_O((\sigma_1;(\sigma_2 \cup \sigma_3))) = M_O(((\sigma_1;\sigma_2) \cup (\sigma_1;\sigma_3)))$
(and similarly for right-distributivity).

c. $M_O((\underline{\text{if}}\ b\ \underline{\text{then}}\ \sigma_1\ \underline{\text{else}}\ \sigma_2\ \underline{\text{fi}};\sigma)) = M_O(\underline{\text{if}}\ b\ \underline{\text{then}}\ (\sigma_1;\sigma)\ \underline{\text{else}}\ (\sigma_2;\sigma)\ \underline{\text{fi}})$.

PROOF. Omitted. □

Remark: Outermost parentheses will often be omitted in the sequel. Also, we omit parentheses in cases such as $\underline{\text{if}}\ b\ \underline{\text{then}}\ \sigma_1;\sigma_2\ \underline{\text{else}}\ \sigma_3 \cup \sigma_4\ \underline{\text{fi}}$, $\mu X[\sigma_1;\sigma_2]$, $\mu Y[\sigma \to \pi]$, etc. Moreover, we write, e.g., $\sigma_1;\sigma_2 \cup \sigma_3$, using the convention that " ; " is assigned higher priority than " $\cup$ ".

LEMMA 3.9. *For each M,*

a. $xM_O(A)y$ *iff one of the following holds:*
- $x \in V_0$, $y \in V_0$, *and* $xC(A)y$
- $x \in V_0$, $\neg\exists z[xC(A)z]$, *and* $y = \perp$
- $x = y = \perp$.

b. *Similar for* $xM_O(X)y$.
c. $M_O(\sigma_1;\sigma_2) = M_O(\sigma_1);M_O(\sigma_2)$.
d. $M_O(\sigma_1 \cup \sigma_2) = M_O(\sigma_1) \cup M_O(\sigma_2)$.
e. $M_O(\underline{\text{if}}\ b\ \underline{\text{then}}\ \sigma_1\ \underline{\text{else}}\ \sigma_2\ \underline{\text{fi}}) = \underline{\text{if}}\ C(b)\ \underline{\text{then}}\ M_O(\sigma_1)\ \underline{\text{else}}\ M_O(\sigma_2)\ \underline{\text{fi}}$.
f. $M_O(\mu X[\sigma]) = M_O(\sigma[\mu X[\sigma]/X])$.

PROOF. Omitted. □

Lemma 3.9 will be the starting point of the definition of denotational semantics, which now follows.

3.3. *Denotational semantics*

Let $M = \langle V,C,E\rangle$, with $V = V_0 \cup \{\bot\}$, C and E as above. In denotational semantics, one directly defines the mapping determined by M, from $\sigma \in S$ to $S \in TR(V)$, without using computation sequences. It is then not immediately clear how to interpret a recursive scheme. The definition for this case - which will turn out to yield the usual least fixed point (though with respect to an unusual partial ordering) - needs some preparation and is, therefore, postponed to section 4. The other cases, for schemes $\sigma \in S$ and $\pi \in P$, are straightforward - for program schemes they should be compared with lemma 3.9 - and now follow:

DEFINITION 3.10 (Denotational semantics). Let $M = \langle V,C,E\rangle$ be as above, $\sigma \in S$ and $\pi \in P$. We define the mappings $M_D(\sigma)$ and $M_D(\pi)$, or $M(\sigma)$ and $M(\pi)$, for short, with $M(\sigma) \in TR(V)$ and $M(\pi) \in HE(V)$, as follows:

1. *Program schemes*

$\sigma \equiv A$: $M(\sigma) = C(A) \cup \{\langle\bot,\bot\rangle\} \cup \{\langle x,\bot\rangle \mid x \in V_0 \wedge \neg\exists y \in V_0[xC(A)y]\}$
$\sigma \equiv X$: $M(\sigma) = E(X) \cup \{\langle\bot,\bot\rangle\} \cup \{\langle x,\bot\rangle \mid x \in V_0 \wedge \neg\exists y \in V_0[xE(X)y]\}$
$\sigma \equiv \sigma_1;\sigma_2$: $M(\sigma) = M(\sigma_1);M(\sigma_2)$
$\sigma \equiv \sigma_1 \cup \sigma_2$: $M(\sigma) = M(\sigma_1) \cup M(\sigma_2)$
$\sigma \equiv \underline{\text{if}}\ b\ \underline{\text{then}}\ \sigma_1\ \underline{\text{else}}\ \sigma_2\ \underline{\text{fi}}$: $M(\sigma) = \underline{\text{if}}\ C(b)\ \underline{\text{then}}\ M(\sigma_1)\ \underline{\text{else}}\ M(\sigma_2)\ \underline{\text{fi}}$
$\sigma \equiv \mu X[\sigma_1]$: postponed.

2. *Boolean schemes*

$\pi \equiv b$: $M(\pi)(x) = C(b)(x)$, $x \in V_0$
$M(\pi)(\bot) = \underline{\text{false}}$

$\pi \equiv Y$: $M(\pi)(x) = E(Y)(x), \quad x \in V_0$
$M(\pi)(\perp) = \underline{\text{false}}$
$\pi \equiv \pi_1 \wedge \pi_2$: $M(\pi) = M(\pi_1) \wedge M(\pi_2)$
$\pi \equiv \sigma \rightarrow \pi_1$: $M(\pi) = M(\sigma) \rightarrow M(\pi_1)$
$\sigma \equiv \underline{\text{if}}\ b\ \underline{\text{then}}\ \pi_1\ \underline{\text{else}}\ \pi_2\ \underline{\text{fi}}$: $M(\pi) = \underline{\text{if}}\ C(b)\ \underline{\text{then}}\ M(\pi_1)\ \underline{\text{else}}\ M(\pi_2)\ \underline{\text{fi}}$
$\pi \equiv \mu Y[\pi_1]$: postponed.

Thus, we see that

- The definition of M applied to constants and variables follows directly from our desire to work with total, extended relations and predicates.
- The definition of M applied to program schemes constructed according to the rules of composition, choice and selection, is the natural one, and in accordance with lemma 3.9.
- The definition of M for boolean schemes is also the expected one - though a corresponce with an operational definition is now not available.
- The definition of M for the two recursive cases is postponed.

4. RECURSION

This section contains the definition of the denotational semantics of recursive schemes. The first subsection brings the introduction of the partial ordering between relations due to Egli, and the usual material on monotonicity, least fixed points of monotonic operators, etc. The second subsection is devoted to the definition proper of the interpretation of recursive schemes, and to the introduction and application of the notion of continuity of an operator.

4.1. *Preliminaries*

DEFINITION 4.1 [Egli] (Ordering between relations).
Let $V = V_0 \cup \{\perp\}$, and let $R,S \in TR(V)$. We define

$$R \leq S \text{ iff } \forall x \begin{cases} \text{if } xR\perp \text{ then } \forall y[xRy \wedge y \neq \perp \rightarrow xSy] \\ \text{if } \neg xR\perp \text{ then } \forall y[xRy \leftrightarrow xSy] \end{cases} .$$

Explanation: We see that $R \leq S$ holds iff, for all x,
- either $xR\perp$ holds - and, possibly, also $xRy_1, xRy_2, \ldots$ - in which case there is a possibility that the information about the computation is not yet complete, and in a better approximation S we may add new outputs, i.e., we always have that $xSy_1, xSy_2, \ldots$ also holds, and we *may* have that $xSz, \ldots$ holds for some new z (and, also, that still $xS\perp$ holds),
- or $\neg xR\perp$ holds. Then the information *is* complete, there are no longer computation sequences which still have to decide about their answer; hence, no additional new outputs are allowed in S, i.e., R and S now coincide on x.

Maybe it is helpful to add here a quotation from [6] as well, where Egli motivates his definition as follows: "... Let us look at the notion of approximating the value of a computation. We think of it as follows: We compute for a certain finite amount of time. If we have not found the value, we approximate by saying that it is $\perp$ at this point. Then we compute further. If we ever find a value, then we know the result. Now let us think of a nondeterministic such computation from a recursive program. Suppose we know all the outcomes along all finite paths of say at most length n. We may then know certain numbers as possible values. Certain paths may not have returned a value. For those we have to compute further. On the other hand, if we have found a number value for every possible path, then we are done. So the point we want to make here is that if a (nonempty) subset of $V_0 \cup \{\perp\}$ approximates the set of outcomes of a program, then either it *is* the outcome of the program or else it has to contain $\perp$ ..." [Last sentence of the quotation slightly adapted, dB.]

A necessary property of "$\leq$" is that it is preserved by the relational operations. This is stated in

LEMMA 4.2. *If* $R \leq S$ *then*
a. $R;T \leq S;T$, *and symmetric*
b. $R \cup T \leq S \cup T$, *and symmetric*
c. if b then R else T fi $\leq$ if b then S else T fi, *and symmetric.*

PROOF. We show only case a. Assume $R \leq S$, and take any x. First assume $xR;T\perp$, and $xR;Ty$ with $y \neq \perp$. Then, for some $z \neq \perp$, $xRz \wedge zTy$. Since $R \leq S$, xSz follows, and, therefore, also xS;Ty. Next assume $\neg xR;T\perp$. Hence $\neg xR\perp$,

and since $R \leq S$, $\forall y[xRy \leftrightarrow xSy]$. Thus, $\forall y[xR;Ty \leftrightarrow xS;Ty]$ follows. □

We also need to define a partial ordering for the predicates. This is straightforward and given in

DEFINITION 4.3 (Ordering between predicates). For $V = V_0 \cup \{\perp\}$, and $p,q \in HE(V)$ we define

$$p \leq q \quad \text{iff} \quad \forall x \in V[p(x) \rightarrow q(x)].$$

(Of course, on the right-hand side "$\rightarrow$" denotes the usual implication from predicate logic.)

Clearly, we have

LEMMA 4.4. *If* $p \leq q$ *then*
a. $p \wedge r \leq q \wedge r$, *and symmetric*
b. $R \rightarrow p \leq R \rightarrow q$
c. $\underline{\text{if}}\ r\ \underline{\text{then}}\ p\ \underline{\text{else}}\ p'\ \underline{\text{fi}} \leq \underline{\text{if}}\ r\ \underline{\text{then}}\ q\ \underline{\text{else}}\ p'\ \underline{\text{fi}}$, *and symmetric.*

PROOF. Clear. □

The next step is the introduction of the notion of *monotonic operators* and their *least fixed points:*

NOTATION 4.5 (Monotonic operators). A *monotonic operator* Φ on $TR(V)$ is a mapping from $TR(V)$ to $TR(V)$ such that $\Phi(R) \leq \Phi(S)$ whenever $R \leq S$. Monotonic operators Ψ on $HE(V)$ are defined similarly.

NOTATION 4.6 (Least fixed points). The *least fixed point* of an operator Φ, denoted by $\mu\Phi$, is a relation with the properties that
a. $\Phi(\mu\Phi) = \mu\Phi$
b. For all R, if $\Phi(R) = R$, then $\mu\Phi \leq R$.
Least fixed points $\mu\Psi$ of operators Ψ are defined similarly.

The question of the existence of least fixed points for operators Φ will be dealt with in section 4.2. The case for the operators Ψ is easier, and given in

LEMMA 4.7. *Let* Ψ *be a monotonic operator over* $HE(V)$. *Then* Ψ *has a least fixed point* $\mu\Psi$.

PROOF. Consider the set $\{p \mid \Psi(p) \leq p\}$. This set is nonempty, since the predicate t, defined by: $t(x) = \underline{\text{true}}$, all $x \in V_0$, and $t(\perp) = \underline{\text{false}}$, is a member of it. Now let us define, for any index set I, $\bigwedge_{i \in I} p_i$ as follows:

$$\left(\bigwedge_{i \in I} p_i\right)(x) = \begin{cases} \underline{\text{true}}, & \text{if } p_i(x) = \underline{\text{true}} \text{ for all } i \in I \\ \underline{\text{false}}, & \text{otherwise.} \end{cases}$$

Then the predicate $\bigwedge\{p \mid \Psi(p) \leq p\}$ has the desired properties of $\mu\Psi$, as follows by a standard application of the Knaster-Tarski argument. (See e.g. [1,2].) □

It should be observed that this proof does not carry over to the Φ's, since neither a greatest element (counterpart of t), nor the operation $\bigwedge_{i \in I}$ are guaranteed to exist.

As a corollary of lemma 4.7 we have

COROLLARY 4.8. *Let* q *be any predicate satisfying* $\Psi(q) \leq q$. *Then* $\mu\Psi \leq q$.

PROOF. Follows from the construction of $\mu\Psi$ in the proof of lemma 4.7. □

4.2. *Denotational semantics of recursive schemes*

Why the interest in least fixed points? Because in a sense to be made precise presently, a recursive scheme $\mu X[\sigma]$ *is* the least fixed point of a certain operator associated with σ.

In order to explain this, we first introduce the notation for these operators which, in turn, needs the definition of a *variant* of an interpretation M.

NOTATION 4.9 (Variants of M). Let $M = \langle V, C, E \rangle$ be as usual, and let $X \in \mathcal{X}$. The interpretation $M\{R/X\}$ is such that $M\{R/X\}(X) = R$, and $M\{R/X\}$ coincides with M for each $A \in \mathcal{A}$, $b \in \mathcal{B}$, $X_1 \in \mathcal{X}$ with $X_1 \neq X$, and $Y \in \mathcal{Y}$. Similar definitions hold for $M\{p/Y\}$. Variants of M can also be used for the operational interpretation, leading to the notation $M\{R/X\}_O(\sigma)$, etc.

NOTATION 4.10 (Operators from schemes). Let $R \in TR(V)$, $X \in X$ and $\sigma \in S$. The operator $\lambda R \cdot \dot{M}\{R/X\}(\sigma)$ maps the element $R \in TR(V)$ to the element $\dot{M}\{R/X\}(\sigma) \in TR(V)$. A similar definition holds for $\lambda p \cdot \dot{M}\{p/Y\}(\pi)$. The meaning of $\lambda R \cdot M\{R/X\}_O(\sigma)$ and of $\lambda p \cdot M\{p/Y\}_O(\pi)$ should also be clear.

We now state - again without full proof - two more lemma's on the operational interpretation, one of a general nature, and the other one providing the central characteristic of recursion:

LEMMA 4.11. *For each* $\sigma,\tau \in S$, $X \in X$, *we have*

$$M_O(\sigma[\tau/X]) = \dot{M}\{M_O(\tau)/X\}_O(\sigma).$$

PROOF. Induction on the structure of σ. □

LEMMA 4.12 (Least fixed point lemma).

$$M_O(\mu X[\sigma]) = \mu[\lambda R \cdot \dot{M}\{R/X\}_O(\sigma)].$$

PROOF.

a. We show that $M_O(\mu X[\sigma])$ *is* a fixed point of $\lambda R \cdot \dot{M}\{R/X\}_O(\sigma)$:
$(\lambda R \cdot \dot{M}\{R/X\}_O(\sigma))(M_O(\mu X[\sigma])) =$
$\dot{M}\{M_O(\mu X[\sigma])/X\}_O(\sigma) =$ (lemma 4.11)
$M_O(\sigma[\mu X[\sigma]/X]) =$ (lemma 3.9f)
$M_O(\mu X[\sigma])$.

b. The proof of: If $\dot{M}\{R/X\}_O(\sigma) = R$, then $M_O(\mu X[\sigma]) \leq R$, is omitted here. It can be given essentially along similar lines as the proof of the main theorem of our paper [4] though the formalism used there is rather different. □

LEMMA 4.12 motivates our next definition, which is the central one of (our treatment of) denotational semantics:

DEFINITION 4.13 (Denotational interpretation of recursive schemes).

1. $M(\mu X[\sigma]) = \mu[\lambda R \cdot \dot{M}\{R/X\}(\sigma)]$
2. $M(\mu Y[\pi]) = \mu[\lambda p \cdot \dot{M}\{p/Y\}(\pi)]$

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

This definition - inspired as it is by lemma 4.12 - seems straightforward. However, we need some additional argument to establish the *existence* of the least fixed points concerned. This we now proceed to do. First we take the second - simpler - case. By lemma 4.7, it is sufficient to show that $\lambda p \cdot M\{p/Y\}(\pi)$ is a monotonic operator:

LEMMA 4.14. *For all* M, *and all* $\pi \in P$, *if* $p \leq q$, *then* $M\{p/Y\}(\pi) \leq M\{q/Y\}(\pi)$.

PROOF. Induction on the structure of π. If $\pi \in B \cup V$, the assertion is clear. If π is of the form $\pi_1 \wedge \pi_2$, $\sigma \to \pi$, or <u>if</u> b <u>then</u> π_1 <u>else</u> π_2 <u>fi</u>, the proof is direct from lemma 4.4. There remains the case that $\pi \equiv \mu Y[\pi_1]$. The argument for this - which is well-known, see e.g. [1] - is the following: We have to show $M\{p/Y\}(\mu Y_1[\pi_1]) \leq M\{q/Y\}(\mu Y_1[\pi_1])$, or, by definition 4.13, that $\mu[\lambda p_1 \cdot M\{p/Y\}\{p_1/Y_1\}(\pi_1)] \leq \mu[\lambda p_1 \cdot M\{q/Y\}\{p_1/Y_1\}(\pi_1)]$. By the proof of lemma 4.7, this is equivalent to showing that
$\bigwedge\{p_1 \mid M\{p/Y\}\{p_1/Y_1\}(\pi_1) \leq p_1\} \leq \bigwedge\{p_1 \mid M\{q/Y\}\{p_1/Y_1\}(\pi_1)\}$, and this inequality follows directly from the induction hypothesis and the definition of $\bigwedge$. □

There remains the justification of the first part of definition 4.13. For this, we need a new property of operators, their *continuity*, which, in turn, uses the notion of *chains* of relations and their least upper bounds (lubs).

NOTATION 4.15 (Chains and their lubs).

a. A chain over $TR(V)$ is a sequence $\{R_i\}_{i=0}^{\infty}$, such that
$R_0 \leq R_1 \leq \ldots \leq R_i \leq \ldots$.

b. The lub of a chain $\{R_i\}_{i=0}^{\infty}$, denoted by $\bigvee_{i=0} R_i$, is a relation such that
(i) $R_j \leq \bigvee_{i=0}^{\infty} R_i$, $j = 0,1,\ldots$
(ii) For all S, if $R_j \leq S$, $j = 0,1,\ldots$, then $\bigvee_{i=0}^{\infty} R_i \leq S$.

Chains *do* have lubs:

LEMMA 4.16. *Each chain* $\{R_i\}_{i=0}^{\infty}$ *has a lub* $\bigvee_{i=0}^{\infty} R_i$.

PROOF. $\bigvee_{i=0}^{\infty} R_i$ is defined as follows: For each x
- either $xR_i\perp$ holds for all $i = 0,1,\ldots$. Then we put, for each $y \in V$, $x(\bigvee_{i=0}^{\infty} R_i)y$ iff xR_iy for some i.

- or $\neg x R_i \perp$ holds for some $i = i_0$. We then put, for each $y \in V$, $x(\bigvee_{i=0}^{\infty} R_i)y$ iff $xR_{i_\infty}y$ for some $i \geq i_0$.
Verification that $\bigvee_{i=0}^{\infty} R_i$ is indeed the lub is left to the reader. □

Remark. It is not true that each two relations R,S have a lub $R \vee S$. In particular, it is not true that $R \cup S$ could be taken as such a ("$\leq$"-) lub.

Next we give the continuity definition.

DEFINITION 4.17 (Continuity). A monotonic operator Φ is called *continuous* iff, for each chain $\{R_i\}_{i=0}^{\infty}$, we have

$$\Phi(\bigvee_{i=0}^{\infty} R_i) = \bigvee_{i=0}^{\infty} \Phi(R_i).$$

The basic relational operations *are* continuous:

LEMMA 4.18. *For* $\{R_i\}_{i=0}^{\infty}$ *a chain,*
a. $(\bigvee_{i=0}^{\infty} R_i);S = \bigvee_{i=0}^{\infty}(R_i;S)$, *and symmetric*
b. $(\bigvee_{i=0}^{\infty} R_i) \cup S = \bigvee_{i=0}^{\infty} (R_i \cup S)$, *and symmetric*
c. $\bigvee_{i=0}^{\infty}$ (<u>if</u> b <u>then</u> R_i <u>else</u> S <u>fi</u>) = <u>if</u> b <u>then</u> $\bigvee_{i=0}^{\infty} R_i$ <u>else</u> S <u>fi</u>, *and symmetric.*

PROOF. Clear from the definitions. □

Caution: Of course, we can also introduce the notion of continuity with respect to the "$\leq$" ordering for predicates. However, the construction rules for boolean schemes do not guarantee continuity. Specifically, it is not, in general, true that, for $\{p_i\}_{i=0}^{\infty}$ a chain, $\bigvee_{i=0}^{\infty} (R \rightarrow p_i) = R \rightarrow \bigvee_{i=0}^{\infty} p_i$. Hence, the results which follow hold *only* for program schemes; for boolean schemes we have monotonic, but not necessarily continuous operators.

Continuous operators allow a nice way of obtaining least fixed points. For this we need

NOTATION 4.19 (Least element for "$\leq$"). Let $O \in TR(V)$ be defined as follows:

$$O = \{\langle x, \perp \rangle \mid x \in V\}.$$

Clearly, $O \leq R$ for all $R \in TR(V)$.

NOTATION 4.20 (Iterating Φ). $\Phi^i(R)$ is defined by: $\Phi^0(R) = R$, $\Phi^{i+1}(R) = \Phi(\Phi^i(R))$.

LEMMA 4.21. *For each continuous* Φ*:*

$$\mu\Phi = \bigvee_{i=0}^{\infty} \Phi^i(O) .$$

PROOF. Clear from the definitions. □

The next lemma asserts that operators derived from program schemes *are* continuous:

LEMMA 4.22. *For each* M, $X \in \mathcal{X}$, $\sigma \in S$, *the operator* $\lambda R \cdot M\{R/X\}(\sigma)$ *is continuous.*

PROOF. We use induction on the structure of σ. The cases that $\sigma \in A \cup \mathcal{X}$, or σ is made up through composition, choice or selection, are clear from the definitions and lemma 4.18. If σ is itself a recursive scheme, $\sigma \equiv \mu X_1[\sigma_1]$, we have, by induction, that for each M, $\lambda R \cdot M\{R_1/X_1\}(\sigma_1)$ is (monotonic and) continuous, hence $\mu[\lambda R_1 \cdot M\{R_1/X_1\}(\sigma_1)]$ exists and can be obtained as $\bigvee_{j=0}^{\infty} S_j$, with $S_0 = O$, $S_{j+1} = M\{S_j/X_1\}(\sigma_1)$. The proof is then completed by showing that, for $\{R_i\}_{i=0}^{\infty}$ a chain, $M\{\bigvee_{i=0}^{\infty} R_i/X\}(\mu X_1[\sigma_1]) = \bigvee_{i=0}^{\infty} M\{R_i/X\}(\mu X_1[\sigma_1])$. The proof of this is - again - essentially the same as given e.g. in [1,2], and omitted here. □

Finally, we state one more lemma, which is the counterpart of lemma 4.11 for denotational interpretations:

LEMMA 4.23.

a. $M(\sigma[\tau/X]) = M\{M(\tau)/X\}(\sigma)$.

b. $M(\pi[\tau/X]) = M\{M(\tau)/X\}(\pi)$.

c. $M(\pi[\pi_1/Y]) = M\{M(\pi_1)/Y\}(\pi)$.

PROOF. Induction on the structure of σ or π. □

We conclude this section with the following

SUMMARY.

1. For each recursive program scheme $\mu X[\sigma]$ we have

$$M(\mu X[\sigma]) = \mu[\lambda R \cdot \dot{M}\{R/X\}(\sigma)] = \bigvee_{i=0}^{\infty} S_i,$$

with $S_0 = \Omega$, $S_{i+1} = \dot{M}\{S_i/X\}(\sigma)$
(this result is justified on the base of the continuity of the operator $\lambda R \cdot \dot{M}\{R/X\}(\sigma)$).

2. For each recursive boolean scheme $\mu Y[\pi]$ we have

$$M(\mu Y[\pi]) = \mu[\lambda p \cdot \dot{M}\{p/Y\}(\pi)]$$

(this result is justified on the base of the monotonicity of the operator $\lambda p \cdot \dot{M}\{p/Y\}(\pi)$).

5. TERMINATION

This section is the central one of our paper. We propose a method of associating with each program scheme σ a boolean scheme π expressing termination of σ. The definition is first motivated, then presented, and finally justified using the tools developed in the previous section.

What do we want to achieve? In order to state our goal, we first give the notation for expressing "proper termination" of a relation:

NOTATION 5.1 (Proper termination of a relation). We define the operation $e: TR(V) \to HE(V)$ by: For each $R \in TR(V)$, and $x \in V$:

$$e(R)(x) \quad \text{iff} \quad \neg xR\perp .$$

Thus, we see that $e(R)(x)$ is true whenever $\perp$ is not a possible outcome of applying R to input x. It should be noted that, by the definition of $TR(V)$, we always have $\perp R \perp$, whence we have $e(R)(\perp) = \underline{\text{false}}$. Here we find the motivation for our choice of $p(\perp) = \underline{\text{false}}$, for any predicate p (notation 3.4). (Of course, this can also be approached more generally: If our domain V were ordered such that $\perp \leq x$ for all $x \in V$, we would want

that, for each p, $p(\bot) \leq p(x)$, and, taking "$\leq$" on W as the implication relation, the choice $p(\bot) =$ false is seen to be the desired one.)

Some properties of the e-operation are stated in

LEMMA 5.2.

a. *If* $R_1 \leq R_2$, *then* $e(R_1) \leq e(R_2)$

b. $e(R_1;R_2) = e(R_1) \wedge (R_1 \rightarrow e(R_2))$

c. $e(R_1 \cup R_2) = e(R_1) \wedge e(R_2)$

d. $e(\underline{\text{if}}\ b\ \underline{\text{then}}\ R_1\ \underline{\text{else}}\ R_2\ \underline{\text{fi}}) = \underline{\text{if}}\ b\ \underline{\text{then}}\ e(R_1)\ \underline{\text{else}}\ e(R_2)\ \underline{\text{fi}}$

e. *For* $\{R_i\}_{i=0}^{\infty}$ *a chain,* $e(\bigvee_{i=0}^{\infty} R_i) = \bigvee_{i=0}^{\infty} e(R_i)$.

PROOF. We prove only case b. We have, for each $x \in V$,

$xR_1;R_2\bot$ iff $xR_1\bot \vee \exists z \neq \bot[xR_1z \wedge zR_2\bot]$.

Hence, $\neg xR_1;R_2\bot$ iff $\neg xR_1\bot \wedge \neg\exists z\neq\bot[xR_1z \wedge zR_2\bot]$

iff $\neg xR_1\bot \wedge \forall z\neq\bot[xR_1z \rightarrow \neg zR_2\bot]$ iff $e(R_1)(x) \wedge (R_1 \rightarrow e(R_2))(x)$. □

We are now sufficiently prepared for the statement of our main problem:

For each program scheme σ, *define a syntactic operation, denoted by* "$\sim$", *say, yielding a boolean scheme* $\tilde{\sigma}$, *such that the following holds:*
For each M:

(5.1) $\qquad M(\tilde{\sigma}) = e(M(\sigma))$.

So what we have to do is:

- Define "$\sim$"
- Show that, when "$\sim$" is used in combination with recursion, the results of section 4 remain valid
- Prove (5.1).

In our justification of the definition of "$\sim$", - and in the remainder of the paper - we shall omit the qualification "proper" in "properly terminating": From now on a terminating computation neither goes on indefinitely, nor aborts on an elementary action being undefined at some intermediate state.

By way of preparation for the definition of "$\sim$", we consider the

various rules of scheme-construction:

- $\sigma \equiv \sigma_1;\sigma_2$: In order that σ terminates for all computations we require that
  . σ_1 terminates for all computations, and
  . σ_2 terminates for all computations which have as input a possible (proper) output of σ_1.
- $\sigma \equiv \sigma_1 \cup \sigma_2$: σ terminates iff both σ_1 and σ_2 terminate.
- $\sigma \equiv$ *if* b *then* σ_1 *else* σ_2 *fi*: This case is clear.
- $\sigma \equiv \mu X[\sigma_1]$. In our explanation of $\tilde{\sigma}$ in this case we use the more intuitive procedure notation already referred to before (comments following definition 2.2). Let *proc* $P;\sigma_1[P]$ be a procedure declaration, i.e., we consider *proc* $P;\underbrace{\ldots P \ldots}_{\sigma_1[P]}$. By the fixed point property $P = \ldots P \ldots$. Applying "$\sim$" on both sides: $\tilde{P} = \sim P \sim \tilde{P} \sim$, which is, informally again, a way of indicating that occurrences of P in $\sigma_1[P]$ lead to occurrences of both P and $\tilde{P}$ in $\sigma_1[P]^{\sim}$ (e.g., $(P;A)^{\sim} = \tilde{P} \wedge (P \rightarrow \tilde{A})$). We, therefore, expect that the boolean scheme we look for is given through the declaration *proc* $Q; \sim P \sim Q \sim$, which is indeed what turns out to be the case.
- We also have to define "$\sim$" for constants and variables. Since these are "atomic", we cannot reduce their termination properties to simpler ones, i.e., for each $A \in \mathcal{A}$ and $X \in \mathcal{X}$, we assume the boolean schemes $\tilde{A} \in \mathcal{B}$ and $\tilde{X} \in \mathcal{Y}$ as given at the outset.

Thus, we can now understand

DEFINITION 5.3 (Syntactic termination operation). For each $\sigma \in \mathcal{S}$, $\tilde{\sigma}$ is an element of $\mathcal{P}$ defined as follows:

- $\sigma \equiv A$: $\tilde{\sigma}$ is some element $\tilde{A}$ in $\mathcal{B}$
- $\sigma \equiv X$: $\tilde{\sigma}$ is some element $\tilde{X}$ in $\mathcal{Y}$
- $\sigma \equiv \sigma_1;\sigma_2$: $\tilde{\sigma} \equiv \tilde{\sigma}_1 \wedge (\sigma_1 \rightarrow \tilde{\sigma}_2)$
- $\sigma \equiv \sigma_1 \cup \sigma_2$: $\tilde{\sigma} \equiv \tilde{\sigma}_1 \wedge \tilde{\sigma}_2$
- $\sigma \equiv$ *if* b *then* σ_1 *else* σ_2 *fi*: $\tilde{\sigma} \equiv$ *if* b *then* $\tilde{\sigma}_1$ *else* $\tilde{\sigma}_2$
- $\sigma \equiv \mu X[\sigma_1]$: $\tilde{\sigma} \equiv \mu\tilde{X}[\tilde{\sigma}_1[\mu X[\sigma_1]/X]]$.

The definition of "$\sim$" having been presented, we next turn to its justification. We precede this with the definition of the notion of a *consistent* interpretation.

NOTATION 5.4. An interpretation M is called consistent iff, for each $A \in \mathcal{A}$ and $X \in \mathcal{X}$, we have $M(\tilde{A}) = e(M(A))$, $M(\tilde{X}) = e(M(X))$.

Thus, through the notion of consistency we guarantee that (5.1) holds for elementary σ.

THEOREM 5.5 (First main theorem). *For each scheme* σ *and interpretation* M*:*
1. *For all* $Y \in \mathcal{Y}$*, and all* p,q *with* $p \leq q$*:* $M\{p/Y\}(\tilde{\sigma}) \leq M\{q/Y\}(\tilde{\sigma})$
2. $M(\tilde{\sigma}) = e(M(\sigma))$*, provided* M *is consistent.*

(Note that the first assertion of the theorem is necessary to justify the definition of $\tilde{\sigma}$ for σ a recursive scheme: We have - implicitly - extended the definition of the class $\mathcal{P}$ with the construction rule: If $\sigma \in \mathcal{S}$, then $\tilde{\sigma} \in \mathcal{P}$. Therefore, we have to verify that this addition preserves monotonicity.)

PROOF. The proof proceeds by simultaneously showing assertions 1 and 2, using induction on the structure of σ. We use the terminology: π is monotonic in $Y_1, Y_2, \ldots$, iff, for $p_1 \leq q_1$, $p_2 \leq q_2, \ldots$, and each M, $M\{p_1/Y_1\}\{p_2/Y_2\} \ldots \{\}(\pi) \leq M\{q_1/Y_1\}\{q_2/Y_2\} \ldots \{\}(\pi)$. (The proof also uses an extension of lemma 4.23b, corresponding to our extension of the construction rules for π. Properly speaking, this extension would have to be taken along as a third assertion in the present proof. However, we have preferred to avoid such further complicating the argument.)

- $\sigma \equiv A$ or $\sigma \equiv X$: Clear from the definitions, in particular because of the consistency condition.
- $\sigma \equiv \sigma_1;\sigma_2$
  1. Assume $p \leq q$. $M\{p/Y\}((\sigma_1;\sigma_2)^{\sim}) = M\{p/Y\}(\tilde{\sigma}_1 \wedge (\sigma_1 \rightarrow \tilde{\sigma}_2))$
     $= M\{p/Y\}(\tilde{\sigma}_1) \wedge (M\{p/Y\}(\sigma_1) \rightarrow M\{p/Y\}(\tilde{\sigma}_2)) \leq$
     $M\{q/Y\}(\tilde{\sigma}_1) \wedge (M\{q/Y\}(\sigma_1) \rightarrow M\{q/Y\}(\tilde{\sigma}_2))$, which follows from the induction hypothesis for $\tilde{\sigma}_1$ and $\tilde{\sigma}_2$, from lemma 4.4, and from the fact that Y does not occur free in σ_1.
  2. $M((\sigma_1;\sigma_2)^{\sim}) = M(\tilde{\sigma}_1 \wedge (\sigma_1 \rightarrow \tilde{\sigma}_2)) = M(\tilde{\sigma}_1) \wedge (M(\sigma_1) \rightarrow M(\tilde{\sigma}_2))$
     $=$ (ind.)$e(M(\sigma_1)) \wedge (M(\sigma_1) \rightarrow e(M(\sigma_2)))$ $=$ (lemma 5.2b)$e(M(\sigma_1);M(\sigma_2))$
     $=$ (def.M)$e(M(\sigma_1;\sigma_2))$.
- $\sigma \equiv \sigma_1 \cup \sigma_2$, or $\sigma \equiv$ <u>if</u> b <u>then</u> σ_1 <u>else</u> σ_2 <u>fi</u>. These cases follow similarly as the previous one using induction and lemma 5.2.

- $\sigma \equiv \mu X[\sigma_1]$. In this part of the proof we use the single symbol μ as an abbreviation for $\mu X[\sigma_1]$.

  1. Assume $p \leq q$. We have $M\{p/Y\}(\mu X[\sigma_1]^{\sim}) = M\{p/Y\}(\mu\tilde{X}[\tilde{\sigma}_1[\mu/X]])$ = (by induction, $\tilde{\sigma}_1$ is monotonic in each $Y \in \mathcal{V}$. Then so is $\tilde{\sigma}_1[\mu/X]$) $\mu[\lambda p_1 \cdot M\{p/Y\}\{p_1/\tilde{X}\}(\tilde{\sigma}_1[\mu/X])] \leq \mu[\lambda p_1 \cdot M\{q/Y\}\{p_1/\tilde{X}\}(\tilde{\sigma}_1[\mu/X])]$, where the last inequality follows in a similar way as in the proof of lemma 4.14.

  2. We show that $M(\mu X[\sigma_1]^{\sim}) = e(M(\mu X[\sigma_1]))$.

By lemma 4.22, we have $M(\mu X[\sigma_1]) = \bigvee_{i=0}^{\infty} S_i$, with $S_0 = o$, $S_{i+1} = M\{S_i/X\}(\sigma_1)$. By definition 5.3, $M(\mu X[\sigma_1]^{\sim}) = M(\mu\tilde{X}[\tilde{\sigma}_1[\mu/X]]) = \mu[\lambda p \cdot M\{p/\tilde{X}\}(\tilde{\sigma}_1[\mu/X])]$ = (lemma 4.23b) $\mu[\lambda p \cdot M\{p/\tilde{X}\}\{M\{p/\tilde{X}\}(\mu)/X\}(\tilde{\sigma}_1)]$ = ($\tilde{X}$ not free in μ) $\mu[\lambda p \cdot M\{p/\tilde{X}\}\{M(\mu)/X\}(\tilde{\sigma}_1)]$. Now let $\Psi \overset{df.}{=} \lambda p \cdot M\{p/\tilde{X}\}\{M(\mu)/X\}(\tilde{\sigma}_1)$. We show that $e(M(\mu))$ is the least fixed point of Ψ.

(i) $e(M(\mu))$ *is* a fixed point of Ψ: $\Psi(e(M(\mu))) = M\{e(M(\mu))/\tilde{X}\}\{M(\mu)/X\}(\tilde{\sigma}_1)$ = (we can use the induction hypothesis, since $M\{e(M(\mu))/\tilde{X}\}\{M(\mu)/X\}$ is consistent) = $e(M\{e(M(\mu))/\tilde{X}\}\{M(\mu)/X\}(\sigma_1))$ = ($\tilde{X}$ not in σ_1) = $e(M\{M(\mu)/X\}(\sigma_1))$ = (lemma 4.23a) = $e(M(\sigma_1[\mu/X]))$ = $e(M(\mu))$, by the fixed point property for the recursive scheme $\mu X[\sigma_1]$.

(ii) We prove that, whenever $\Psi(p) \leq p$, then $e(M(\mu)) \leq p$. (Note the use of corollary 4.8.) So assume $\Psi(p) \leq p$. To show $e(M(\mu)) \leq p$, i.e., $e(\bigvee_i S_i) \leq p$, with S_i as above. By continuity of e (lemma 5.2a), $e(\bigvee_i S_i) = \bigvee_i e(S_i)$. Thus, it is sufficient to show $e(S_i) \leq p$ for all i. Clearly, $e(S_0) \leq p$. Now assume

$$e(S_i) \leq p. \tag{5.2}$$

We also have, by definition,

$$S_i \leq M(\mu). \tag{5.3}$$

We now show that $e(M\{S_i/X\}(\sigma_1)) \leq p$, or, equivalently, since $\tilde{X}$ does not occur free in σ_1, $e(M\{S_i/X\}\{e(S_i)/\tilde{X}\}(\sigma_1)) \leq p$, or, by the induction hypothesis, that $M\{S_i/X\}\{e(S_i)/\tilde{X}\}(\tilde{\sigma}_1) \leq p$, where we have used the consistency of $M\{S_i/X\}\{e(S_i)/\tilde{X}\}$. Furthermore, we have that $\Psi(p) \leq p$, i.e. that

(5.4) $\quad M\{p/\tilde{X}\}\{M(\mu)/X\}(\tilde{\sigma}_1) \leq p.$

The desired result: $M\{e(S_i)/\tilde{X}\}\{S_i/X\}(\tilde{\sigma}_1) \leq p$ now follows, using (5.2), (5.3), (5.4) and monotonicity. □

This completes the proof of the first main theorem of our paper.

6. DERIVATIVES

This section is devoted to a comparison of our method of dealing with program termination as presented in section 5, and the approach of HITCHCOCK & PARK [7] using the notions of well-founded relation, and of derivative of a program scheme. In particular, we extend the main theorem of [7] to nondeterministic programs, and we give a new proof of it using our method.

A number of new notations are first introduced.

NOTATION 6.1 (Special constant schemes).

a. Ω and Δ are two program scheme constants with the convention that, for all M,

$M(\Omega) = O \overset{df.}{=\!=} \{\langle x,\perp\rangle \mid x \in V\}$

$M(\Delta) = I \overset{df.}{=\!=} \{\langle x,x\rangle \mid x \in V\}$

b. ω and θ are two boolean scheme constants with the convention that, for all M,

$M(\omega) = f$, where $f(x) = \underline{\text{false}}$, for all $x \in V$

$M(\theta) = t$, where $\begin{cases} t(x) = \underline{\text{true}}, \text{ all } x \in V_0 \\ t(\perp) = \underline{\text{false}}. \end{cases}$

NOTATION 6.2 (Extended construction rules for schemes).
The following construction rules for schemes are added to the rules of definition 2.2:

1. *Program schemes*
   a. If $\sigma \in S$ and $\pi \in P$ then $\sigma \cup \pi \in S$.
2. *Boolean schemes*
   a. If $\pi_1, \pi_2 \in P$ then $\pi_1 \vee \pi_2 \in P$
   b. If $\pi \in P$ then $\bar{\pi} \in P$
   c. If $\sigma \in S$ and $\pi \in P$ then $\sigma \circ \pi \in P$.

NOTATION 6.3 (Additional operations on relations and predicates).
For $S \in TR(V)$, $p, p_1, p_2 \in HE(V)$ we define
a. $x(S \cup p)y$ iff $xSy \vee p(x)$
b. $(p_1 \vee p_2)(x)$ iff $p_1(x) \vee p_2(x)$
c. $\bar{p}(x)$ iff $\neg p(x)$, for all $x \in V_0$; $\bar{p}(\perp) = \underline{\text{false}}$
d. $(S \circ p)(x)$ iff $\exists y[xSy \wedge p(y)]$.

Notations 6.2 and 6.3 are linked through:

NOTATION 6.4 (Extended definition of M).
For each M we define
a. $M(\sigma \cup \pi) = M(\sigma) \cup M(\pi)$
b. $M(\pi_1 \vee \pi_2) = M(\pi_1) \vee M(\pi_2)$
c. $M(\bar{\pi}) = \overline{M(\pi)}$
d. $M(\sigma \circ \pi) = M(\sigma) \circ M(\pi)$.

Remarks.

1. As will be seen in the sequel, our use of complementation is such that the monotonicity of those operators for which we need the existence of their least fixed points, is not disturbed.
2. It is easily seen that the relationship between "$\circ$" and "$\rightarrow$" operations is the following: $S \rightarrow p = \overline{S \circ \bar{p}}$.
3. From the definitions it follows that $(R \circ t)(x)$ iff $\exists y \neq \perp[xRy]$.
4. The construction rule leading to schemes of the form $\sigma \cup \pi$ is somewhat ad hoc, and included only to make direct translation of the Hitchcock & Park formalism into ours possible. A more general approach would be to embed P into S, essentially through the convention that each $p \in HE(V)$ determines a $P \in TR(V)$ as follows: xPy iff $p(x) \wedge (x = y)$.

The class of simple schemes to be defined next is actually somewhat smaller than the class with the same name of [7], where a form of relational concatenation is also allowed. This construction rule could be incorporated without too much trouble, however. It should be noted that simple schemes do not allow "iterated" recursion, i.e., no constructs of the form $\mu X_1[\ldots \mu X_2[\ldots]\ldots]$. A remark on this restriction follows at the end of the present section.

DEFINITION 6.5 (Simple schemes). A program scheme σ is X-simple iff it is constructed according to the following rules:

a. $\sigma \equiv X$

b. $\sigma \equiv A$, for some $A \in \mathcal{A}$

c. $\sigma \equiv \sigma_1;\sigma_2$, with σ_1,σ_2 both X-simple

d. $\sigma \equiv \sigma_1 \cup \sigma_2$, with σ_1,σ_2 both X-simple

e. $\sigma \equiv \underline{\text{if}}\ b\ \underline{\text{then}}\ \sigma_1\ \underline{\text{else}}\ \sigma_2\ \underline{\text{fi}}$, with σ_1,σ_2 both X-simple.

For X-simple schemes, Hitchcock & Park define the notion of uppper- and lower derivative (with respect to X, this being silently understood in the remainder of this section).

DEFINITION 6.6 (Derivatives). For an X-simple scheme σ, the upper derivative $\dot{\sigma}$ - yielding an element of S -, and the lower derivative $\dot{\underline{\sigma}}$- yielding an element of P - are defined as follows:

1. Upper derivative:

$\sigma \equiv X$: $\dot{\sigma} \equiv \Delta$

$\sigma \equiv A$: $\dot{\sigma} \equiv \Omega$

$\sigma \equiv \sigma_1;\sigma_2$: $\dot{\sigma} \equiv \dot{\sigma}_1 \cup (\sigma_1;\dot{\sigma}_2)$

$\sigma \equiv \sigma_1 \cup \sigma_2$: $\dot{\sigma} \equiv \dot{\sigma}_1 \cup \dot{\sigma}_2$

$\sigma \equiv \underline{\text{if}}\ b\ \underline{\text{then}}\ \sigma_1\ \underline{\text{else}}\ \sigma_2\ \underline{\text{fi}}$: $\dot{\sigma} \equiv \underline{\text{if}}\ b\ \underline{\text{then}}\ \dot{\sigma}_2\ \underline{\text{else}}\ \dot{\sigma}_2\ \text{fi}.$

2. Lower derivative

$\sigma \equiv X$: $\dot{\underline{\sigma}} \equiv \omega$

$\sigma \equiv A$: $\dot{\underline{\sigma}} \equiv \overline{\tilde{A}}$

$\sigma \equiv \sigma_1;\sigma_2$: $\dot{\underline{\sigma}} \equiv \dot{\underline{\sigma}}_1 \vee (\sigma_1 \circ \dot{\underline{\sigma}}_2)$

$\sigma \equiv \sigma_1 \cup \sigma_2$: $\dot{\underline{\sigma}} \equiv \dot{\underline{\sigma}}_1 \vee \dot{\underline{\sigma}}_2$

$\sigma \equiv \underline{\text{if}}\ b\ \underline{\text{then}}\ \sigma_2\ \underline{\text{else}}\ \sigma_2\ \underline{\text{fi}}$: $\dot{\underline{\sigma}} \equiv \underline{\text{if}}\ b\ \underline{\text{then}}\ \dot{\underline{\sigma}}_1\ \underline{\text{else}}\ \dot{\underline{\sigma}}_2.$

NOTATION 6.7 (Derivatives "in a recursive scheme"). For σ an X-simple scheme, we write

$\mathring{\sigma} \overset{\text{df.}}{=\!=} \dot{\sigma}[\mu X[\sigma]/X]$

$\mathring{\underline{\sigma}} \overset{\text{df.}}{=\!=} \dot{\underline{\sigma}}[\mu X[\sigma]/X].$

Next, we present another important tool in the approach of [7]:

DEFINITION 6.8 (Well-founded relations). Let $R \in TR(V)$. R is called well-founded in an element $x \in V$ iff $x \neq \perp$ and there does not exist an infinite sequence $x_0 = x, x_1, x_2, \ldots$, all $x_i \in V_0$, such that $x_i R x_{i+1}$, $i = 0,1,\ldots$.

It is possible to connect the notions of well-foundedness and of least fixed point of an operator:

LEMMA 6.9. The two assertions
1. R is well-founded in $x \in V$
2. $\mu[\lambda p \cdot \overline{R \circ \overline{p}}](x)$ holds
are equivalent.

PROOF. See [7].

Remarks.
1. Note that $\overline{R \circ \overline{p}}$ is monotonic - though not necessarily continuous [7] - in p.
2. There is a slight difference with the approach in [7] in that "$\perp$" does not play a part in that paper. However, it may be verified that the argument for lemma 6.9 remains valid.

We shall use lemma 6.9 in its alternative form:

COROLLARY 6.10. R *is well-founded in* x *iff* $\mu[\lambda p \cdot (R \rightarrow p)](x)$ *holds.*

PROOF. Lemma 6.9 and a remark after notation 6.4. □

Three further pieces of notation are introduced:

NOTATION 6.11 (Equality of schemes under some (all) interpretation(s)).
a. For program schemes σ, τ and interpretation M, we write $\models_M \sigma = \tau$ iff $M(\sigma) = M(\tau)$.
b. We write $\models \sigma = \tau$ whenever $\models_M \sigma = \tau$ holds for all *consistent* M.

NOTATION 6.12 (Well-founded part of a scheme). For $\sigma \in S$, we write

$$\iota(\sigma) \overset{\text{df.}}{=} \mu Y[\sigma \rightarrow Y]$$

where Y is some boolean variable not occurring free in σ.

The main theorem of Hitchcock & Park's approach to termination - in our version, extended for nondeterminacy - is

THEOREM 6.13. *For each X-simple program scheme* σ

$$\models \mu X[\sigma]^{\sim} = \iota(\overset{\circ}{\sigma} \cup \underset{\circ}{\sigma}).$$

PROOF. We present a proof using the tools developed sofar; we organize the proof in a number of lemma's:

LEMMA 6.14. *For* σ *an X-simple scheme:*

$$\models \tilde{\sigma} = (\dot{\sigma} \to \tilde{X}) \wedge \bar{\underset{\cdot}{\sigma}} .$$

PROOF. Induction on the structure of σ. Take any consistent M.

a. $\sigma \equiv X$. We have to show
$M(\tilde{X}) = M((\Delta \to \tilde{X}) \wedge \bar{\omega})$, or
$M(\tilde{X}) = (M(\Delta) \to M(\tilde{X})) \wedge M(\bar{\omega})$, or
$M(\tilde{X}) = (I \to M(\tilde{X})) \wedge \bar{f}$, or
$M(\tilde{X}) = M(\tilde{X}) \cap t$.

b. $\sigma \equiv A$. We have to show
$M(\tilde{A}) = M((\Omega \to \tilde{X}) \wedge \bar{\bar{A}})$, or
$M(\tilde{A}) = (M(\Omega) \to M(\tilde{X})) \wedge M(\bar{\bar{A}})$, or
$M(\tilde{A}) = (O \to M(\tilde{X})) \wedge M(\tilde{A})$, or
$M(\tilde{A}) = t \wedge M(\tilde{A})$.

c. $\sigma \equiv \sigma_1;\sigma_2$. We have to show
$\models (\sigma_1;\sigma_2)^{\sim} = ((\sigma_1;\sigma_2)^{\cdot} \to \tilde{X}) \wedge \bar{(\sigma_1;\sigma_2)_{\cdot}}$. We rewrite the right-hand side:
$\models (\sigma_1;\sigma_2)^{\cdot} \to \tilde{X}) \wedge \bar{(\sigma_1;\sigma_2)_{\cdot}} =$ (def. of $\dot{\sigma}, \underset{\cdot}{\sigma}$)
$((\dot{\sigma}_1 \cup \sigma_1;\dot{\sigma}_2) \to \tilde{X}) \wedge (\underset{\cdot}{\sigma}_1 \vee \sigma_1 \circ \underset{\cdot}{\sigma}_2)^{-} =$ (see (*) below)
$(\dot{\sigma}_1 \to \tilde{X}) \wedge ((\sigma_1;\dot{\sigma}_2) \to \tilde{X}) \wedge \bar{\underset{\cdot}{\sigma}}_1 \wedge \overline{\sigma_1 \circ \underset{\cdot}{\sigma}_2} =$ (see (**) below)
$(\dot{\sigma}_1 \to \tilde{X}) \wedge (\sigma_1 \to (\dot{\sigma}_2 \to \tilde{X})) \wedge \bar{\underset{\cdot}{\sigma}}_1 \wedge (\sigma_1 \to \bar{\underset{\cdot}{\sigma}}_2) =$ (see (***) below)
$(\dot{\sigma}_1 \to \tilde{X}) \wedge \bar{\underset{\cdot}{\sigma}}_1 \wedge (\sigma_1 \to ((\dot{\sigma}_2 \to \tilde{X}) \wedge \bar{\underset{\cdot}{\sigma}}_2)) =$ (ind. hyp.)
$\tilde{\sigma}_1 \wedge (\sigma_1 \to \tilde{\sigma}_2) =$ (def. "$\sim$")
$(\sigma_1;\sigma_2)^{\sim}$.

where

(*) : $(R_1 \cup R_2) \to p = (R_1 \to p) \wedge (R_2 \to p)$

(**) : $(R_1;R_2) \to p = R_1 \to (R_2 \to p)$

(***): For each R: $(R \to p) \wedge (R \to q) = (R \to p \wedge q)$.

d. $\sigma \equiv \sigma_1 \cup \sigma_2$; We have

$\models \tilde{\sigma} = (\sigma_1 \cup \sigma_2)^{\sim} = \tilde{\sigma}_1 \wedge \tilde{\sigma}_2 =$ (ind.)

$(\dot{\sigma}_1 \to \tilde{X}) \wedge \bar{\dot{\sigma}}_1 \wedge (\dot{\sigma}_2 \to \tilde{X}) \wedge \bar{\dot{\sigma}}_2 =$

$(\dot{\sigma}_1 \cup \dot{\sigma}_2 \to \tilde{X}) \wedge (\dot{\sigma}_1 \vee \dot{\sigma}_2)^{-} = ((\sigma_1 \cup \sigma_2)^{\cdot} \to \tilde{X}) \wedge (\sigma_1 \cup \sigma_2)^{-}_{\cdot} =$

$(\dot{\sigma} \to \tilde{X}) \wedge \bar{\dot{\sigma}}$.

e. $\sigma \equiv$ <u>if</u> b <u>then</u> σ_1 <u>else</u> σ_2 <u>fi</u>. Straightforward by induction. □

LEMMA 6.15. *For* σ *an X-simple scheme*

$$\models \tilde{\sigma}[\mu X[\sigma]/X] = (\mathring{\sigma} \to \tilde{X}) \wedge \bar{\mathring{\sigma}} .$$

PROOF. Direct from the previous lemma and notation 6.7. □

LEMMA 6.16. *For* σ *an X-simple scheme*

$$\models \mu\tilde{X}[\tilde{\sigma}[\mu X[\sigma]/X]] = \mu\tilde{X}[(\mathring{\sigma} \to \tilde{X}) \wedge \bar{\mathring{\sigma}}] .$$

PROOF. Lemma 6.15. □

LEMMA 6.17. *For each relation* R *and predicate* q*:*

$$\mu[\lambda p\cdot((R \to p) \cap \bar{q})] = \mu[\lambda p\cdot((R \cup q) \to p)].$$

PROOF. Call the left-hand side ℓ and the right-hand side r. We show that $r \leq \ell$, leaving the other half of the proof to the reader. By corollary 4.8, it is sufficient to show: $((R \cup q) \to \ell) \leq \ell$, or, by the fixed point property, $((R \cup q) \to \ell) \leq (R \to \ell) \wedge \bar{q}$. First we show $((R \cup q) \to \ell) \leq \bar{q}$. Take any x, and assume that $((R \cup q) \to \ell)(x)$ holds, but that $\bar{q}(x)$ does not hold, i.e., $q(x)$ holds. Then, since $((R \cup q) \to \ell)(x)$ by assumption, from $q(x)$ we conclude that $\ell(x)$ holds, whence, by the fixed point property of ℓ, $\bar{q}(x)$ holds. Contradiction. Next to show $((R \cup q) \to \ell) \leq (R \to \ell)$. This follows immediately from the definition of "$\to$". □

Finally, we obtain our conclusion: For σ an X-simple scheme:

$$\models \mu X[\sigma]^{\sim} = \iota(\mathring{\sigma} \cup \mathring{\sigma}). \tag{6.1}$$

This is shown as follows: Applying lemma 6.17, we get

$$\models \mu\tilde{X}[(\overset{\circ}{\sigma} \to \tilde{X}) \wedge \overset{\circ}{\tau}] = \mu\tilde{X}[(\overset{\circ}{\sigma} \cup \overset{\circ}{\tau}) \to \tilde{X}].$$

Combining this with lemma 6.16, and using the definition of "$\sim$" and ι, we obtain (6.1), as was to be shown. □

This completes our discussion of the relationship between the approach of Hitchcock & Park, and ours, in sofar as X-simple schemes are concerned. In [7], a sketch is also given of a way of extending the main theorem to systems of (simultaneously) recursive procedures. A comparison of this with our formalism would necessitate a replacement of our use of iterated recursion with that of simultaneous recursion. This having been performed, the additional argument to establish the analogue of (6.1) for systems does not require any essential new considerations, reason why we prefer to leave this problem to the interested reader.

REFERENCES

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

[2] DE BAKKER, J.W., *The fixed point approach in semantics: theory and applications*, in J.W. de Bakker (ed.), Foundations of Computer Science, pp.3-53, Mathematical Centre Tracts 63, Amsterdam (1975).

[3] DE BAKKER, J.W., *Flow of control in the proof theory of structured programming*, Proc. 16th IEEE Symp. on Foundations of Computer Science (1975).

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

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

[6] EGLI, H., *A mathematical model for nondeterministic computations*, ETH Zürich (1975).

[7] HITCHCOCK, P. & D. PARK, *Induction rules and proofs of termination*, in Automata, Languages and Programming (M. Nivat, ed.), p.225-251, North-Holland, Amsterdam (1973).

[8] MANNA, Z., *Mathematical Theory of Computation*, McGraw-Hill (1974).

[9] DE ROEVER, W.P., *Recursive Program Schemes: Semantics and Proof Theory*, Ph.D. Thesis, Free University, Amsterdam (1975).

[10] SCOTT, D., *Outline of a mathematical theory of computation*, Proc. of the Fourth Annual Princeton Conference on Information Sciences and Systems, pp.170-176 (1970).

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