stichting
mathematisch
centrum

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 135/80 APRIL

R.J. BACK

SEMANTICS OF UNBOUNDED NONDETERMINISM

Preprint

2e boerhaavestraat 49 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).

1980 Mathematics subject classification: 68B10, 68B05

ACM-Computing Review-categories: 5.24

Semantics of unbounded nondeterminism[*)]

by

R.J. Back

ABSTRACT

A program construct is proposed, for which the assumption of bounded nondeterminism is not natural. It is shown that the simple approach of taking the powerdomain of the flat cpo does not produce a correct semantics for programs in which nondeterminism is unbounded. The powerdomain approach is then extended to computation paths, resulting in an essentially operational semantics for programs of unbounded nondeterminism.

KEY WORDS & PHRASES: *unbounded nondeterminism, powerdomains, denotational semantics, nondeterministic assignment statements, termination*

[*)] This report will appear in the proceedings of ICALP 80, Springer Lecture Notes in Computer Science.

1. INTRODUCTION

Nondeterminism is usually introduced into a programming language in the form of a new control structure. One possibility is to define a binary construct, S_1 *or* S_2, which has the effect of selecting either S_1 or S_2 (but not both) for execution. The choice between the two alternatives is made nondeterministically. Another possibility, introduced by DIJKSTRA [76], is to generalise the conditional statement. The effect of the guarded command *if* $B_1 \rightarrow S_1 \square \ldots \square B_n \rightarrow S_n$ *fi* is to execute some statement S_i for which the corresponding guard B_i is true. Nondeterminism is possible in this case, because the guards are not required to be mutually exclusive.

There is, however, another way in which nondeterminism can be introduced into a sequential programming language. This is by allowing the basic statements to be nondeterministic. This can be achieved by generalising the ordinary assignment statement to a *nondeterministic assignment statement*. Such a construct has been used in HAREL [77] and in BACK [78], and in a somewhat different form, in BAUER [77]. A nondeterministic assignment statement has the form

$$x: = x'.Q$$

and has the effect of assigning to x some value x' that satisfies the condition Q, in which both x and x' (and also other variables) may occur free. Nondeterminism can occur, because there may be more than one x' that satisfies the condition Q for a given value of x. In this case some x' satisfying Q is choosen nondeterministically, and assigned to x. If no x' satisfies Q, then the effect of the statement is undefined.

This latter form of nondeterminism is actually very common, although in a somewhat disguised form. Consider a program S which calls a procedure p, and assume that p is specified by giving the entry and exit conditions for the procedure. Usually we will try to understand the way in which S works by only considering the information about p given in the specification of this procedure. The exit condition, however, may not define a unique final result of calling p. In understanding S, we then have to consider all possible final states which can result from the call, i.e.

we will in effect be looking upon S as a program with an nondeterministic basic statement p. The nondeterminism here results from lack of information as to the effect of calling p, rather than from the fact that S is executed by a nondeterministic machine. Somebody observing only the working of S and knowing only the specification of p, cannot, however, tell the difference between these two views. (One could argue that in the first case, because the procedure p actually is executed by a deterministic mechanism, the same result will always be choosen for the same initial state. On the other hand, it is possible that a nondeterministic mechanism executing p would choose to give the same result for the same initial state in all observed calls, but still could give some other result for some future call on p.)

We will here consider a simple iterative language embodying this kind of nondeterminism. For this purpose, let Var be a nonempty set of variables. Assume that certain function, predicate and constant symbols are given, and let Form be the set of all first-order formulas built out of variables and these symbols. We will let x,y and z range over variables and B,P,Q,R range over first-order formulas in Form.

Let Stat be the set of program statements, recursively defined by

$$S ::= \quad x{:=}x'.Q \mid S_1;S_2 \mid \textit{if } B \textit{ then } S_1 \textit{ else } S_2 \textit{ fi} \mid \textit{while } B \textit{ do } S_1 \textit{ od}.$$

Here S, S_1 and S_2 range over program statements. The effect of the first construct was already explained above. The other constructs have their usual meaning of composition, selection and iteration.

Our purpose here is to discuss how the semantics of programming languages such as Stat, containing nondeterministic basic statements, is to be defined. The semantics of nondeterministic control structures have been successfully defined in PLOTKIN [76]. This definition, however, makes essential use of the assumption that the nondeterminism is *bounded*. This means that execution of a program component for a given initial state either can only produce a finite number of different results, or then it must be possible that execution never terminates. The possibility that execution for some initial state will be guaranteed to terminate and at the same time may produce infinitely many different results is thus excluded.

The assumption of bounded nondeterminacy, also made and discussed in DIJKSTRA [76], is intuitively justified for nondeterministic control structures such as those given above. In these cases the choice of how to proceed is made between a finite number of alternatives. The set of all possible executions of a program from a given initial state will therefore form a finitely branching tree. If this tree contains infinitely many terminal nodes then the tree itself must be infinite. By Königs lemma, this means that there must be an infinite branch in the tree, i.e. an infinite execution starting from the given initial state is possible.

There does not, however, seem to be any intuitive reason for not considering *basic statements* of unbounded nondeterminacy. In his book Dijkstra e.g. rejects the basic statement "set x to any positive integer", because it cannot be implemented using guarded commands. While this is true, there is on the other hand nothing wrong intuitively in calling a procedure p, with the specification

proc p; *entry* true; *exit* $x > 0$;

This corresponds to using the nondeterministic assignment statement

$x:=x.(x > 0)$

in the program. The fact that this statement cannot be implemented is of no concern here, the purpose of the statement is only to express the information available about the effect of calling the procedure. Moreover, from the point of understanding a program, there is no difference between basic statements of bounded and unbounded nondeterminacy, both seem to be equally well defined and easy to understand. (This same point has also been made by BOOM [78]).

The nondeterministic assignment statement is more powerful that the nondeterministic control statements given above, because nondeterministic control structures can be simulated using nondeterministic assignment statements, while the converse does not necessarily hold. Thus e.g. the binary choice construct S_1 *or* S_2 can be expressed in our language by the statement

$c:=c.(c=1 \text{ or } c=2)$; *if* $c=1$ *then* S_1 *else* S_2 *fi*.

The assignment statement can also be expressed using the nondeterministic assignment statement. The assignment statement $x:=t$ is expressed by the nondeterministic assignment

$x := x'.(x' = t)$.

E.g. $x := x+1$ corresponds to the statement $x := x'.(x' = x+1)$.

An obvious generalisation of this nondeterministic assignment statement is to allow simultaneous assignment to several variables, i.e. we would allow nondeterministic assignment statements of the form

$x_1,\dots,x_n:=x_1',\dots,x_n'.Q$.

We will only treat the single variable form below, for reasons of simplicity.

2. FUNCTIONAL SEMANTICS BASED ON $P(\Sigma_\perp)$

We will start by defining the semantics of Stat using the powerdomain $P(\Sigma_\perp)$ introduced in PLOTKIN [76].

Let D be a nonempty set, serving as the domain of interpretation for the formulas in Form. The set of proper program states is defined to be $\Sigma = \text{Var} \to D$, while $\Sigma_\perp = \Sigma \cup \{\perp\}$ is the set of all program states, including the *undefined state* $\perp$. An ordering of approximation is defined in $\Sigma_\perp$ as usually, by the condition

$s \sqsubseteq s'$ iff $s = \perp$ or $s = s'$,

for any $s, s' \in \Sigma_\perp$. It is easily shown that $\Sigma_\perp$ is a complete partial order (cpo) with respect to this ordering. The undefined state is used to indicate nontermination, as usual.

The meaning of a nondeterministic program will be a function from the initial states to the set of possible final states for the given initial state. We therefore need the powerset of $\Sigma_\perp$. Let us define the set of possible results

$$P(\Sigma_\perp) = \{A \subset \Sigma_\perp \mid A \neq \emptyset\}.$$

A subset A of $\Sigma_\perp$ is *bounded*, if $|A| < \infty$ or $\perp \in A$ ($|A|$ is the cardinality of A). The set of possible bounded results is

$$P_B(\Sigma_\perp) = \{A \subset \Sigma_\perp | A \neq \emptyset \text{ and } A \text{ is bounded}\}.$$

An ordering of approximation is defined between elements of $P(\Sigma_\perp)$ as follows:

$$A \sqsubseteq A' \text{ iff } \forall s\in A.\exists s'\in A'.s \sqsubseteq s' \text{ and } \forall s'\in A'.\exists s\in A.s \sqsubseteq s',$$

for any A, A' $\in P(\Sigma_\perp)$. An equivalent formulation for $P(\Sigma_\perp)$ is

$$\begin{aligned} A \sqsubseteq A' \text{ iff either } & \perp \in A \text{ and } A - \{\perp\} \subset A' \\ \text{or } & \perp \notin A \text{ and } A = A'. \end{aligned}$$

We then have that both $P(\Sigma_\perp)$ and $P_B(\Sigma_\perp)$ are cpo's with respect to this ordering.

We define the set of state transformations by $M(\Sigma_\perp) = \Sigma \to P(\Sigma_\perp)$ and the set of bounded state transformations by $M_B(\Sigma_\perp) = \Sigma \to P_B(\Sigma_\perp)$.

An ordering of approximation between state transformations is defined by

$$m \sqsubseteq m' \text{ iff } m(s) \sqsubseteq m'(s) \text{ for all } s \in \Sigma,$$

for any m, m' $\in M(\Sigma_\perp)$. We then have that $M(\Sigma_\perp)$ and $M_B(\Sigma_\perp)$ are both cpo's with respect to this ordering.

To define the meaning of the statements in Stat, we will also need the set of truth values T = {tt,ff} and the set of predicates on Σ, $W(\Sigma) = \Sigma \to T$.

Let $m \in M(\Sigma_\perp)$. The extension of m to $m^\dagger:P(\Sigma_\perp) \to P(\Sigma_\perp)$ is defined as follows. First, let $m'\colon \Sigma_\perp \to P(\Sigma_\perp)$ be defined by

$$m'(s) = \begin{cases} m(s), & \text{if } s \in \Sigma \\ \{s\}, & \text{if } s = \perp . \end{cases}$$

Then $m^\dagger$ is defined by

$$m^\dagger(A) = \cup\{m'(s) \mid s \in A\}.$$

The same construction extends $m \in M_B(\Sigma_\perp)$ to $m^\dagger:P_B(\Sigma_\perp) \to P_B(\Sigma_\perp)$.

Composition and selection is now easy to define. Let m and m' be elements of $M(\Sigma_\perp)$, and let b be an element of $W(\Sigma)$. The composition m;m' of m and m', defined by

$$(m;m')(s) = m'^\dagger(m(s)), \text{ for } s \in \Sigma,$$

will then be an element of $M(\Sigma_\perp)$. Similarly, the selection of m or m' by b, $(b \to m,m')$, defined by

$$(b \rightarrow m,m')(s) = \begin{cases} m(s), & \text{if } b(s) = tt \\ m'(s), & \text{if } b(s) = ff \end{cases}$$

will also be an element of $M(\Sigma_\perp)$.

The same construction defines composition and selection in $M_B(\Sigma_\perp)$; $M_B(\Sigma_\perp)$ will also be closed with respect to these operations. Composition and selection will be monotonic in both $M(\Sigma_\perp)$ and $M_B(\Sigma_\perp)$. In $M_B(\Sigma_\perp)$ these operations will also be continuous. However, composition in $M(\Sigma_\perp)$ is not continuous. This is the main technical reason for requiring bounded nondeterminacy, i.e. the requirement is needed to guarantee continuity of the control structures used.

We will finally define the iteration operator. Let Δ and Ω be two special elements of $M(\Sigma_\perp)$, defined for any $s \in \Sigma$ by $\Delta(s) = \{s\}$ and $\Omega(s) = \{\perp\}$. Let $b \in W(\Sigma)$ and let $m \in M(\Sigma_\perp)$. The approximates $(b*m)^n \in M(\Sigma_\perp)$ are then defined for $n = 0,1,2,\ldots$ by

$$(b*m)^0 = \Omega$$
$$(b*m)^{n+1} = (b \rightarrow m;(b*m)^n,\ \Delta),\ n = 0,1,\ldots \quad .$$

Because composition and selection is monotonic, it is easily proved that $(b*m)^0 \sqsubseteq (b*m)^1 \sqsubseteq (b*m)^2 \sqsubseteq \ldots$, so the least upper bound of this sequence exists, as $M(\Sigma_\perp)$ is a cpo. Thus we may define the iteration of m while b by

$$(b * m) = \bigsqcup_{n=0}^{\infty} (b * m)^n.$$

Before giving the meanings of the statements in Stat, we give one last technical definition. Let $s \in \Sigma$. Then $s[d/x] \in \Sigma$, defined by

$$s[d/x](y) = \begin{cases} d, & \text{if } x = y \\ s(y), & \text{otherwise.} \end{cases}$$

Let first $\mathcal{W}$: Form $\rightarrow$ $W(\Sigma)$ be a function that assigns a predicate on Σ to each formula of Form. The function $\mathcal{W}$ is defined using the interpretation function for the predicate, function and constant symbols of Form. The definition is omitted here.

The meaning of unbounded nondeterministic statements could now be given by the function $\mathcal{M}$:Stat $\rightarrow M(\Sigma_\perp)$, defined as follows:

(i) $\mathcal{M}(x{:=}x'.Q)(s) = \begin{cases} \{s[d/x] \mid d \in D_s\}, & \text{if } D_s \neq \emptyset \\ \{\perp\}, \text{ if } D_s = \emptyset. \end{cases}$

where $D_s = \{d \in D \mid \mathcal{W}(Q)(s[d/x']) = tt\}$

(ii) $\mathcal{M}(S_1;S_2) = \mathcal{M}(S_1);\mathcal{M}(S_2)$

(iii) $\mathcal{M}($*if* B *then* S_1 *else* S_2 *fi*$) = (\mathcal{W}(B) \rightarrow \mathcal{M}(S_1),\mathcal{M}(S_2)]$

(iv) $\mathcal{M}($*while* B *do* S_1 *od*$) = (\mathcal{W}(B) * \mathcal{M}(S_1))$.

If the nondeterministic assignment statements $x{:=}x'.Q$ are restricted in a way which guarantees that only a finite number of values x' satisfying Q exist for any state $s \in \Sigma$ (i.e. $|D_s| < \infty$), then $\mathcal{M}(x{:=}x'.Q)(s) \in P_B(\Sigma_\perp)$. In this case the range of $\mathcal{M}$ is actually $M_B(\Sigma_\perp)$, and $\mathcal{M}$ gives the intuitively correct meaning to statements in Stat. If, however, no such restrictions are made, then the interpretation $\mathcal{M}$ is counterintuitive, as we will show in the next section.

3. WEAK AND STRONG TERMINATION

The problem with the meaning function M, defined above, is that it does not treat termination correctly. To see the problem, we will consider an example taken from DIJKSTRA [76,p.77]. Let S be the statement

$$S: \textit{while } x \neq 0 \textit{ do if } x \geq 0 \textit{ then } x := x-1$$
$$\textit{else } x := x.(x \geq 0) \textit{ fi od} \; .$$

Let S_1 denote the body of this loop.

Intuitively, this program must terminate for any initial value of x, be it positive, zero or negative. However, for negative initial values of x, the meaning function M says that termination is not guaranteed. To see this, let us compute

$$M(\textit{while } x \neq 0 \textit{ do } S_1 \textit{ od})(-1).$$

For simplicity, we here identify the state with the value of x in the state (we may assume that Var = {x}).

Let us denote $b = W(x \neq 0)$ and $m_1 = M(S_1)$. We have that $b(x) = tt$ iff $x \neq 0$ and

$$m_1(x) = \begin{cases} \{x-1\} & , x \geq 0 \\ N & , x < 0, \end{cases}$$

where $N = \{0, 1, 2, \ldots\}$. Let further $m^i = (b * m_1)^i$, for $i = 0,1,\ldots$.

We have that for $x \geq 0$,

$$m^i(x) = \begin{cases} \{0\}, & x < i \\ \{\bot\}, & x \geq i \end{cases} \; .$$

Using this, we compute

$$\begin{aligned} m^0(-1) &= \Omega(-1) = \{\bot\} \\ m^1(-1) &= (b \to m_1;m^0, \Delta)(-1) = m_1;m^0(-1) = m^{0\dagger}(N) = \{\bot\} \\ m^2(-1) &= (b \to m_1;m^1, \Delta)(-1) = m_1;m^1(-1) = m^{1\dagger}(N) = \{0,\bot\} \\ m^3(-1) &= (b \to m_1;m^2, \Delta)(-1) = m_1;m^2(-1) = m^{2\dagger}(N) = \{0,\bot\},\ldots \end{aligned}$$

We get that $m^i(-1) = \{0,\bot\}$, for $i \geq 2$. Consequently,

$$M(S)(-1) = \bigsqcup_{i=0}^{\infty} m^i(-1) = \{0,\bot\} \; .$$

Thus $M(S)(-1)$ contains $\bot$, stating that the loop S is not guaranteed to terminate for $x = -1$ initially. This contradicts our intuition about the behaviour of the loop S for this initial state.

The meaning function M actually formalises *strong termination* of while loops, instead of the usual, intuitive notion of termination. A loop

$$\textit{while } B \textit{ do } S_1 \textit{ od}$$

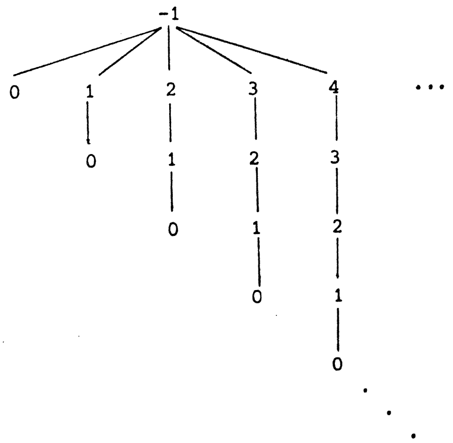
is said to be strongly terminating if for each initial state s there is an integer n_s such that the loop is guaranteed to terminate in less than n_s iterations

(DIJKSTRA [78]). We will call termination that is not strong *weak termination*.

We therefore have to reinterpret the meaning of the undefined state. We have that $\perp \in M(S)(s)$ iff either it is possible that S does not terminate for the initial state s, or that S is guaranteed to terminate for s, but that the termination is weak. Thus the meaning function M does not distinguish between possible nontermination and weak termination.

Termination is always strong when the nondeterminism of a program is bounded. This is again a consequence of Königs lemma. The set of all executions of a program from some given initial state will then form a finitely branching tree. If the program is guaranteed to terminate for the given initial state, then each branch will be finite. By Königs lemma, this means that the tree itself must be finite, and therefore there is only a finite number of different branches in the tree. Thus there is a maximum number of iteration that any branch needs before termination, i.e. the termination is strong.

However, when we allow the branching of the execution tree to be infinite, Königs lemma is no longer applicable. I.e. it is possible that each branch of the tree is finite but that no longest branch exists in the tree, because there are infinitely many branches. An example is provided by the execution tree of the example treated above, for the initial state x = -1:



The failure of the powerdomain $P(\Sigma_\perp)$ to capture the correct notion of termination in the presence of unbounded nondeterminacy can thus be explained by noting that it is built on an erroneous inference: The fact that there after any number of iterations of a loop still could be unfinished computations going on, does not justify the conclusion that there could be a nonterminating computation of the loop.

One might hope that the right notion of termination could be captured by changing the approximation ordering, without changing $P(\Sigma_\perp)$. However, the following program will give the same sequence of approximations $\{0,\perp\}$ for x=-1 as the previous program, but will not be guaranteed to terminate:

while $x \neq 0$ *do* $x:=x.(x=0 \text{ or } x=1)$ *od*

We therefore conclude that the set $P(\Sigma_\perp)$ does not give enough information to decide between weak termination and nontermination.

The condition that we try to capture is that no branch in the execution tree of a program, for a given initial state, is infinitely expanded. This again means that we have to distinguish between different unfinished execution paths in the approximates of the loop, i.e. we are essentially forced into an operational semantics. This will be the subject of the next section.

4. OPERATIONAL SEMANTICS BASED ON $H(\Sigma_\omega)$

We will here show how the approach to nondeterminacy based on $P(\Sigma_\perp)$, explained above, can be adapted to provide a semantic definition for programs of unbounded nondeterminacy. Basically the adaption consists in considering sets of sequences of states in stead of just sets of states as is done in $P(\Sigma_\perp)$. The fact that we use sequences of states in defining the semantics of our programming language, where the sequences roughly correspond to the execution sequence of programs, is the reason for calling the semantic definition operational.

Let Σ as before be the set of states. We will have three different kinds of *execution paths*:

(i) *Terminal paths,* which are sequences of the form $<s_1,\ldots,s_n>$, where $n \geq 1$ and $s_i \in \Sigma$ for $i = 1,\ldots,n$.

(ii) *Unfinished paths*, which are sequences of the form $<s_1,\ldots,s_n,\perp>$, where $\perp$ is a special bottom element not in Σ, $n \geq 0$ and $s_i \in \Sigma$ for $i = 1,\ldots,n$.

(iii) *Infinite paths*, which are infinite sequences of the form $<s_1,s_2,\ldots>$, where $s_i \in \Sigma$ for $i = 1, 2, \ldots$.

The set of all execution paths will be denoted Σ_ω.

Intuitively, a terminal path corresponds to an execution which has terminated, an infinite path corresponds to a nonterminating execution and unfinished paths correspond to executions which have not been completed. The bottom element is used to indicate that the path in question can be extended, by continuing the execution.

An approximation relation is defined in Σ_ω as follows: For paths h and h' in Σ_ω,

$$h \sqsubseteq h' \text{ iff either } h \text{ is terminal or infinite and } h = h', \\ \text{or } h \text{ is unfinished and } \bar{h} < h'.$$

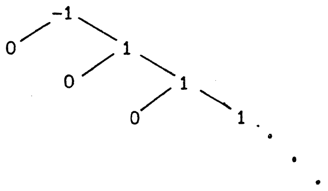
Here $\bar{h}$ denotes the path h with the possible trailing element $\perp$ deleted. We use the notation $h \leq h'$ to express that h is an initial segment of h'($h < h'$ when h is a proper initial segment of h').

LEMMA 1. Σ_ω is a cpo with respect to $\sqsubseteq$.□ (Proofs of theorems and lemmas are given ~~in~~ BACK [79].)

The meaning of a nondeterministic program S will be a function $N(S)$, which assigns to each initial state $s \in \Sigma$ the set of all possible execution paths, by which the execution can continue from s. As an example, consider the program

S': *while* $x \neq 0$ *do* $x := x.(x = 0 \text{ or } x = 1)$ *od*.

The execution tree of this program, for initial state $x = -1$, is



Thus we have that

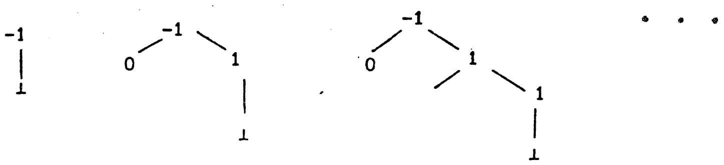
$N(S')(-1) = \{<0>, <1,0>, <1,1,0> ,..., <1,1,1,...>\}$

This set contains, besides all finite paths of form <1,1,...,0> also the infinite path <1,1,...>, reflecting the fact that execution of the program does not necessarily terminate.

The set $N(S')(-1)$ will actually be computed as the limit of an approximation sequence. The elements of this sequence are formed by performing only a certain number of iterations and then aborting the computation. Thus we get the set $N(S)(-1)$ as the limit of the sequence

$H_0 = \{<\perp>\}$
$H_1 = \{<0>, <1,\perp>\}$
$H_2 = \{<0>, <1,0>, <1,1,\perp>\}$.
...

This corresponds to the sequence of execution trees below:



The next tree in the sequence is constructed by replacing the bottom element at the end of the unfinished path by the two possible successor states. The new unfinished path is then marked as such, by adding the bottom element to it.

The idea of constructing a new execution tree from another tree by extending some unfinished branches of the tree underlies the notion of approximation between sets of execution paths. For two sets of execution paths, H and H', approximation is defined in the same way as in $P(\Sigma_\perp)$, i.e.

$$H \sqsubseteq H' \text{ iff } \forall h \in H . \exists h' \in H'.\ h \sqsubseteq h' \text{ and }$$
$$\forall h' \in H' . \exists h \in H.\ h \sqsubseteq h'.$$

It turns out, however, that the sets of execution paths do not form a complete partial order under this ordering relation. In fact, they are not even partially ordered by the approximation relation defined. In order to get a cpo, we will need to put some restrictions on the sets of execution paths allowed.

The appropriate restrictions can be found by considering the way in which the execution trees are constructed. We start from an initial tree which only contains the path $<\perp>$. This tree is then extended step by step, by extending each unfinished branch of the tree by all its immediate successor nodes (of which there might be a finite or an infinite number). In this way we construct the finite approximations of the executions tree. Finally the execution tree itself is constructed by taking the limits of all paths in the finite approximations. In other words, if there is a growing path sequence $h_0 \sqsubseteq h_1 \sqsubseteq h_2 \sqsubseteq \ldots$ in the finite approximations, then the limit must contain the least upper bound of this path sequence. Conversely, any path in the limit must be the least upper bound of some growing path sequence in the finite approximations.

Any set of execution paths corresponding to an execution tree constructed in this manner must satisfy the following three requirements. First, the set cannot be empty. This is because the initial execution tree has the execution path $<\perp>$, and all other execution trees are constructed by extending this unfinished path.

A second property shared by all sets of execution paths generated in this way is *flatness*. This is defined as follows: A set H of execution paths is *flat* if $\bar{h} \leq h' \Rightarrow h = h'$ holds for any two paths h and h' in H. Thus, if H is flat and $h = <s_1,s_2, \ldots>$ and $h' = <s_1',s_2',\ldots>$ are two execution paths in H, then $s_i \neq s_i'$ for some $i \geq 1$, where both s_i and s_i' are elements of Σ. This is a consequence of the way in which unfinished paths are extended. The new paths created by extending an unfinished path are all different, because they have different last states.

The third property shared by all sets of execution paths generated by nondeterministic programs is *closedness*. A set H of execution paths is said to be *closed*, if the following holds: Let $h_0 \sqsubseteq h_1 \sqsubseteq h_2 \sqsubseteq \ldots$ be a sequence of unfinished paths of unbounded length (i.e. there is no upper bound of the lengths of the paths in the sequence). Assume that for each h_i is this sequence, there is some path h_i' in H such that $h_i \sqsubseteq h_i'$. Then the infinite path $\sqcup h_i = h$ belongs to the set H. This property is a consequence of the way in which the limit of the sequence of finite approximations is constructed: In the sequence of finite approximations of H there must be a sequence of unfinished paths of unbounded lengths growing along the path h. Otherwise the paths h_i' in H could not be constructed. But this means that the least upper bound of this sequence of unfinished paths, which also is h, must belong to the set H.

Let us now define the set $H(\Sigma_\omega)$ by

$$H(\Sigma_\omega) = \{H \subseteq \Sigma_\omega \mid H \text{ is nonempty, flat and closed}\}.$$

We then have the following result:

THEOREM 1. $H(\Sigma_\omega)$ is a cpo with respect to the ordering $\sqsubseteq$. The least upper bound of a sequence $H_0 \sqsubseteq H_1 \sqsubseteq H_2 \sqsubseteq \ldots$ of elements in $H(\Sigma_\omega)$ is

$$\bigsqcup_{i=0} H_i = \{ \bigsqcup_{i=0} h_i \mid h_i \in H_i,\ i = 0,1,2, \ldots \text{ and } h_0 \sqsubseteq h_1 \sqsubseteq h_2 \sqsubseteq \ldots \}\ . \ \square$$

$H(\Sigma_\omega)$ will now be taken as the set corresponding to $P(\Sigma_\bot)$. Analogous with the treatment of $P(\Sigma_\bot)$, we introduce the set $N(\Sigma_\omega) = \Sigma \to H(\Sigma_\omega)$, in which approximation is defined in the same way as in $M(\Sigma_\bot)$, i.e.

$$n \sqsubseteq n' \text{ iff } n(s) \sqsubseteq n'(s) \text{ for every } s \in \Sigma,$$

for $n, n' \in N(\Sigma_\omega)$. As before, $N(\Sigma_\omega)$ will be a cpo with respect to this ordering.

Continuing as before, we define the extension of $n: \Sigma \to H(\Sigma_\omega)$ to $n^\dagger: H(\Sigma_\omega) \to H(\Sigma_\omega)$. Let $n': \Sigma_\omega \to H(\Sigma_\omega)$ be defined for $h \in \Sigma_\omega$ by

$$n'(h) = \begin{cases} \{h \cdot h' \mid h' \in n(\mathrm{last}(h))\}, & \text{if } h \text{ is terminal} \\ \{h\} & \text{otherwise} \end{cases}$$

Here $h \cdot h'$ denotes the sequence h concatenated with the sequence h'. We then define

$$n^\dagger(H) = \cup \{n'(h) \mid h \in H\},$$

for $H \in H(\Sigma_\omega)$. The fact that $n^\dagger$ is well-defined is established by the lemma:

LEMMA 2. For any $n \in N(\Sigma_\omega)$, if $H \in H(\Sigma_\omega)$, then $n^\dagger(H) \in H(\Sigma_\omega)$. $\square$

Composition and selection in $N(\Sigma_\omega)$ is then defined as before, i.e.

$$(n_1;n_2)(s) = n_2^\dagger(n_1(s)), \text{ for } s \in \Sigma, \text{ and}$$

$$(b \to n_1,n_2)(s) = \begin{cases} n_1(s), & \text{if } b(s) = tt \\ n_2(s), & \text{if } b(s) = ff \end{cases} \quad \text{for } s \in \Sigma.$$

LEMMA 3. Composition and selection is monotonic in $N(\Sigma_\omega)$. $\square$

Let Δ' and Ω' be two elements in $N(\Sigma_\omega)$, defined by

$$\Delta'(s) = \{<s>\} \quad , \text{ for each } s \in \Sigma, \text{ and}$$
$$\Omega'(s) = \{<\bot>\} \quad , \text{ for each } s \in \Sigma.$$

Here $<s>$ denotes the sequence with s as the only element.

Let $b \in W(\Sigma)$ and $n \in N(\Sigma_\omega)$. We then define $(b * n)$ as before. First, let

$$(b * n)^0 = \Omega', \text{ and}$$
$$(b * n)^{i+1} = (b \to n;(b * n)^i, \Delta'), \text{ for } i = 0,1,2,\ldots \ .$$

As before, $(b * n)^0 \sqsubseteq (b * n)^1 \sqsubseteq \ldots$ follows from the monotonicity of composition and selection. Iteration is then defined as

$$(b * n) = \bigsqcup_{i=0}^{\infty} (b * n)^i.$$

We are now ready to define the semantics of unbounded nondeterministic statements.

We assume that the function $\mathcal{W}$ is given as before. The meaning of statements in Stat is then given by the function $\mathcal{N}$:Stat → $N(\Sigma_\omega)$, defined as follows:

(i) $\mathcal{N}(x{:=}x'.Q)(s) = \begin{cases} \{<s[d/x]> \mid d \in D_s\} & \text{, if } D_s \neq \emptyset \\ \{<\bot>\} & \text{, if } D_s = \emptyset \end{cases}$

where $D_s = \{d \in D \mid \mathcal{W}(Q)(s[d/x']) = tt\}$.

(ii) $\mathcal{N}(S_1;S_2) = \mathcal{N}(S_1);\mathcal{N}(S_2)$

(iii) $\mathcal{N}(\textit{if } B \textit{ then } S_1 \textit{ else } S_2 \textit{ fi}) = (\mathcal{W}(B) \rightarrow \mathcal{N}(S_1),\mathcal{N}(S_2))$

(iv) $\mathcal{N}(\textit{while } B \textit{ do } S_1 \textit{ od}) = (\mathcal{W}(B) * \mathcal{N}(S_1))$

It is easy to check that this definition does give the correct semantics for the example program in the previous section, i.e. the definition agrees with our intuition, treating both strong and weak termination as proper termination.

The domain $N(\Sigma_\omega)$ can also be used for defining the semantics of recursive programs. In order to do this, we require the following theorem.

<u>THEOREM 2</u>. Composition and selection is continuous in $N(\Sigma_\omega)$. □

Recursion can be introduced into our language by defining a new set Svar of statement variables, and adding two new productions to the recursive definition of statements:

$$S ::= \ldots X \mid \mu X.S \ .$$

Here $\mu X.S$ has the effect of executing S, with X recursively bound to S, i.e. any call on X is replaced with the execution of the statement S (X is a statement variable).

To defined the semantics of the recursive statements, we need environments $E = \text{Svar} \rightarrow N(\Sigma_\omega)$. The meaning function will now be of the type $\mathcal{N}'$: Stat → $(E \rightarrow N(\Sigma_\omega))$. The semantic equations are then the following:

(i) $\mathcal{N}'(x{:=}x'.Q)(\eta)(s) = \mathcal{N}(x{:=}x'.Q)(s)$

(ii) $\mathcal{N}'(S_1;S_2)(\eta) = \mathcal{N}'(S_1)(\eta);\mathcal{N}'(S_2)(\eta)$

(iii) $\mathcal{N}'(\textit{if } B \textit{ then } S_1 \textit{ else } S_2 \textit{ fi})(\eta) = (\mathcal{W}(B) \rightarrow \mathcal{N}'(S_1)(\eta),\ \mathcal{N}'(S_2)(\eta))$

(iv) $\mathcal{N}'(\textit{while } B \textit{ do } S_1 \textit{ od})(\eta) = (\mathcal{W}(B) * \mathcal{N}'(S_1)(\eta))$

(v) $\mathcal{N}'(X)(\eta) = \eta(X)$

(vi) $\mathcal{N}'(\mu X.S)(\eta) = \mu n.\mathcal{N}'(S_1)(\eta[n/X])$.

Here η ranges over elements of E. The notation $\eta[n/X]$ means the environment η with the value at X changed to $n \in N(\Sigma_\omega)$. The existence of the least fixed point in (vi) is guaranteed by theorem 2.

A domain similar to $H(\Sigma_\omega)$, using trees and the powerset ordering by PLOTKIN [76] has also been discussed in FRANCEZ [79], with the aim of providing a denotational semantics for nondeterministic, communicating sequential processes. Another, somewhat similar approach, has also been made by KOSINSKI [78], who is concerned with defining the denotational semantics of data flow languages.

6. SUMMARY

The mathematical semantics of nondeterministic programs has been defined in

PLOTKIN [76] using powerdomains. This definition, however, only works when the nondeterminism of the program is bounded. We have above argued that unbounded nondeterminism, introduced by a nondeterministic assignment statement, is a meaningful construct in a programming language. We have shown how to define the semantics of programs with unbounded nondeterminism using an extension of Plotkins construction $P(\Sigma_\omega)$. This extension considers sets of sequences of states (execution sequences) instead of just sets of states. This provides a richer structure, in which it is possible to give the correct semantics of unbounded nondeterminism. It was shown that the sets of execution sequences form a complete partially ordered set, provided that we restrict ourselves to sets which are nonempty, flat and closed. The reasonableness of these assumptions was also shown.

ACKNOWLEDGEMENT. I would like to thank Jaco de Bakker for pointing out the problem with unbounded nondeterminism to me, and Edger W. Dijkstra for his explanation of the difference between weak and strong termination. I am grateful to Lambert Meertens, David Park, Gordon Plotkin, Maarel Karttunen and Ari de Bruin for discussions on this subject.

REFERENCES

[78] BACK, R.J.R., *On the correctness of refinement steps in program development,* Dept. of Computer Science, Univ. of Helsinki, Report A-1978-4.

[79] BACK, R.J.R., *Semantics of unbounded nondeterminism,* Computing Centre of Univ. of Helsinki, Res. Rep. No 8, 1979.

[77] DE BAKKER, J.W., *Semantics of infinite processes using generalised trees,* Math. Centrum Report IW 82/77.

[78] BAUER, F.L., *Design of a programming language for a program transformation system,* 1-8. Jahrestagung, Informatik Fachbereich 16, Springer Verlag.

[78] BOOM, H.J., *A weaker precondition for loops,* Mathematisch Centrum report IW 104/78.

[76] DIJKSTRA, E.W., *A discipline of programming,* Prentice-Hall, 1976.

[78] DIJKSTRA, E.W., *Private communication.*

[79] FRANCEZ, N., HOARE, C.A.R., LEHMAN, D.J. & DE ROEUER, W.P., *Semantics of nondeterminism, concurrency and communication,* Journal of Computer and System Sciences, Vol. 19, No. 3, December 1979, pp. 290-308.

[77] HAREL, D., PNUELI, A. & STAVL, J., *A complete axiomatic system for proving deductions about recursive programs,* Proc. 9th annual ACM Symp. on the Theory of Computing, Boulder, Colorado, May 1977.

[78] KOSINSKI, P.R., *A straightforward denotational semantics for nondeterminant data flow programs,* 5th Annual ACM Symposium on Principles of *Programming languages,* Tucson, January 1978.

[76] PLOTKIN, G.D., *A powerdomain construction,* SIAM J. of Computing 5, 3, September 1976.