COMPUTER PROGRAM SEMANTICS IN SPACE AND TIME

by N. G. de Bruijn

1. INTRODUCTION.

This note can be considered as an addition to a ten year old note [1]. We aim at a new treatment of recursive procedures, i.e. a new version of section 11 of [1]. This new treatment also effects the other sections, but there the alterations that have to be made are quite obvious.

In [1] we used a state space Ω that was extended to a space Ω^+ by adding a single element ∞. This element ∞ played a role in the semantics only: it is not something that can be referred to in the programs. Its semantic role is to indicate non-termination.

In the present note we follow a system that handles some further information, i.e. something corresponding to runtime. Where the system of [1] only distinguished between runtime being finite or infinite, the present system might be able to say exactly what the runtime is in cases where it is finite.

For practical applications it might be interesting to develop runtime administration for terminating programs, but this was not the main motivation for this study. The reason was rather of a theoretical nature. The semantic treatment of [1] (sect.11) turned out to be hard to combine with fixed point semantics. Mr.R.Wieringa, who implemented a large part of [1] in AUTOMAT-, had to introduce a new notion of orde between predicates and had to impose slightly awkward monotonicity restrictions in order to establish a correspondence between the

recursion semantics of [1] and fixed point theory. The system proposed in this note is much easier in this respect. Yet, if we weaken the runtime information by distinguishing between finite and infinite runtime only, the semantics can be expected to be the one of [1].

Our present system takes care of runtime by describing a relation between the moment t where the execution of a program starts, and the moment t' where the execution ends. Semantical information about a program will have the form of a predicate on $(\Omega \times T) \times (\Omega \times T)$ (whereas in [1] it had the form of a predicate on $\Omega^+ \times \Omega^+$. It is quite reasonable to think of predicates in which the relation between t and t' is expressed in a form like

$$t + f(\omega,\omega') \leq t' \leq t + g(\omega,\omega'),$$

and it is also quite reasonable to choose the semantics of primitive program constructs in accordance with this form. Nevertheless we shall not set it as a rule that our predicates should necessarily have this form. What we do require, however, is that $t \leq t'$ is somehow enforced, and we shall realize this by restricting our predicates to the set of all those quadruples (ω, t, ω', t') for which $t \leq t'$.

The interpretation of t and t' is obvious. In a case where $t < \infty$, $t' = \infty$ we of course say that the program does not terminate. In cases where $t = \infty$, $t' = \infty$ we can say that the program execution never started, since some other program that had to be executed first, took infinitely long.

In order to be able to say that the sum of the lengths of infinitely many time intervals is infinite, we restrict ourselves to time moments which are either integers or the symbol ∞.

Just like in [1], where programs never explicitely referred to the element ∞, in our present system the programs will usually not refer to t in any way, except for the program 'delay'.

In this note we shall discuss the relation between t and t' only as far as it is relevant for the treatment of recursion. In

fact we take the point of view that no program takes time, except for the "overhead" of a procedure call. This overhead is connected with the fake program we call "delay", which takes time without doing anything else: its semantics may be described by $(\omega = \omega') \wedge (t' = t+1)$. One might say that every case of non-termination is caused by an infinity of executions of "delay", in spite of the fact that other program components might try to make it worse.

Another point of view in which this note differs from [1] is a simplification: we have given up the "relativistic" attitude (see section 3).

Part of the philosophy underlying the note [1] was that semantical discussion can be kept on a mathematical level without entering into the syntax of a programming language. The correctness of the code in which the program is presented to a computer is a matter we can reduce to the question of the correctness of a compiler. This is not essentially different from the problem of the correctness of the translation of programs in higher order languages.

The philosophy that semantics can be built up without entering into syntax, is elaborated in section 8 of this paper.

2. Predicates on the set Λ.

We start from a set Ω (which is called 'state space'). And we introduce the 'time space' T, defined as

$$T = \mathbb{Z} \cup \{\infty\}$$

(where $\mathbb{Z}$ is the set of integers and ∞ is a new element). In T we have addition and order. The ordinary addition of $\mathbb{Z}$ is extended by $\infty + t = t + \infty = \infty$ for all $t \in T$, and the ordinary order of $\mathbb{Z}$ is extended by agreeing that $k < \infty$ for all $k \in \mathbb{Z}$.

The set Λ is defined as

$\Lambda = \{ (\omega, t, \omega', t') \mid \omega \in \Omega, t \in T, \omega' \in \Omega t' \in T, t \leq t' \}$.

Given a point (ω, t, ω', t'), we may refer to the pair (ω, t) as 'initial', and to the pair (ω', t') as "final".

We write $Pred(\Lambda)$ for the collection of all predicates on Λ. Particular predicates are (i) "TRUE", which is identically true on Λ, (ii) "FALSE", which is identically false on Λ, (iii)"NONTERM" (for non-termination) which has as its value the proposition $t' = \infty$, (iv) a predicate which we shall denote by J, given by

$$J(\omega, t, \omega', t') = (t=t') \wedge ((t' < \infty) \rightarrow (\omega = \omega')). \qquad (1)$$

It is inconvenient to admit all arbitrary predicates on Λ, for in cases where $t' = \infty$ the value of ω' has no sensible interpretation, and for our semantics it would be awkward to make the distinction between different final pairs (ω', ∞). (Of course it neither makes sence to distinguish between different initial pairs (ω, ∞), but there it causes no trouble for our semantic system). Somehow we want to consider all the (ω', ∞)'s as equal, but we do not want to loose the cartesian product structure of our space. Therefore we shall restrict predicates to being"constant at infinity".

We say that a predicate $P \in Pred(\Lambda)$ <u>is constant at infinity</u> if $P(\omega, t, \omega', \infty) = P(\omega, t, \omega_2', \infty)$ for all $\omega, \omega', \omega_2' \in \Omega$ and all $t \in T$. The set of all $P \in Pred(\Lambda)$ which are constant at infinity will be called $Pred^*(\Lambda)$.

For every $P \in Pred(\Lambda)$ we construct a $P^* \in Pred^*(\Lambda)$ as follows: $P^*(\omega, t, \omega', t') = P(\omega, t, \omega', t')$ if $t' < \infty$, and

$$P^*(\omega, t, \omega', \infty) = \exists_{\rho \in \Omega} P(\omega, t, \rho, \infty)$$

for all $\omega, \omega' \in \Omega$, $t \in T$. If P is itself in $Pred^*(\Lambda)$ already, we obviously have $P^* = P$. This applies in particular to the examples TRUE, FALSE, NONTERM and J, mentioned above.

We shall write $P \subset Q$ for predicate implication (rather than using the dubious notation $P \rightarrow Q$), so $P \subset Q$ means

$$\forall_{(\omega, t, \omega', t')} P(\omega, t, \omega', t') \rightarrow Q(\omega, t, \omega', t')$$

And we use the sign $=$ for equivalence of predicates (and we shall often just call it equality), so $P = Q$ means $(P \subset Q) \wedge (Q \subset P)$.

The notation $P \subset Q$ helps us to remind that the set of points satisfying P is a subset of the set of points satisfying Q. Quite often we interpret a predicate as an amount of information, and then we have to keep in mind that $P \subset Q$ does not mean that Q gives more information than P. It is just the other way around: P gives all the information presented by Q, but possibly more.

With the above notation P^*, Q^* we obviously have

$$(P \subset Q) \rightarrow (P^* \subset Q^*)$$

for all $P, Q \in Pred(\Lambda)$.

Quite often we want to describe a predicate by means of some expression E containing ω, t, ω', t'. We shall use the notation λE in order to denote the predicate P for which $P(\omega, t, \omega', t') = E$ for all ω, t, ω', t'. And if P is obtained from E this way, we write $\lambda^* E$ for the P^* corresponding to P. As examples we mention

$$\lambda^*((\omega = \omega') \wedge (t' = \infty)) = \lambda^*(t' = \infty) = \lambda(t' = \infty) = \text{NONTERM},$$

$$\lambda^*((\omega = \omega') \wedge (t = t')) = \lambda((\omega = \omega') \wedge (t = t')) \vee (t = t' = \infty)) =$$

$$= \lambda((t = t') \wedge ((t' < \infty) \rightarrow (\omega = \omega'))) = J.$$

If P and Q are in $Pred(\Lambda)$ we define the 'boolean matrix product', or 'boolean convolution', denoted by $P * Q$, as follows:

$$P * Q = \lambda \exists_{\sigma \in \Omega, s \in T} (P(\omega, t, \sigma, s) \wedge Q(\sigma, s, \omega', t'))$$

We note that this convolution is associative:

$$(P * Q) * R = P * (Q * R).$$

It is not hard to show that for all $P \in Pred(\Lambda)$

$$P * J = P^*, \qquad (2)$$

and therefore $(P * Q)^* = P * Q * J = P * Q^*$. In particular, if Q is constant at infinity, then $P * Q$ is constant at infinity.

As a special case of (2) we mention

$$J * J = J^* = J.$$

3. Semantic information.

We assume that we have a set called Prog(Ω). The elements of this set are called programs (or "programs on Ω"). And we assume to have a mapping "Totinf" (which stands for "total information"), mapping Prog(Ω) into Pred*(Λ). The interpretation is that if π is a program, then Totinf(π) is a predicate on Λ that presents all the semantic information about π. That is to say in terms of operational interpretation: there is an execution of π leading from initial state (ω, t) to final state (ω', t') if and only if the quadruple (ω, t, ω', t') satisfies the predicate Totinf(π).

Quite often it happens that for a given program π it is hard to find Totinf(π) and, moreover, this total information is not always important in all its details. We can usually be quite happy with something that is simpler and weaker. That means that we work with some R $\in$ Pred*(Λ) such that

$$\text{Totinf}(\pi) \subset R. \qquad (1)$$

In most cases we start from the other end. We have some predicate R as our goal, and we want to find a program π such that (1) holds. Then we say that R is a program specification, and that π is a program that satisfies the specification.

In [1] we tried to avoid the introduction of a thing like Totinf. The idea behind this is that it might be useful to have a semantic system in which people with different ideas about the total information are still able to communicate about things they do

agree on, and also that it leaves some freedom to those who implement a programming language. The idea of [13], if extended to our present system, means that we work with a predicate w on $Prog(\Omega) \times Pred^*(\Lambda)$, with the interpretation that $w(\pi, P)$ expresses that $P(\omega, t, \omega', t')$ is true if (but not necessarily "only if") (ω, t, ω', t') presents the initial and final states of some execution. The connection with Totinf is

$$\text{if} \quad Totinf(\pi) \subset P \quad \text{then} \quad w(\pi, P)$$

for all $\pi \in Prog(\Omega)$, $P \in Pred^*(\Lambda)$.

In the present note we shall not follow the line of [13]. The advantage of avoiding $Totinf(\pi)$ might not compensate the disadvantage that the basic properties of $w(\pi, P)$ are harder to formulate than those of $Totinf(\pi)$. There is an analogy in topology, where there is a possibility ("point-free topology") to restrict the discussions to "open sets" as basic objects, without bothering whether these objects are sets (of "points") indeed. The price that has to be paid for this "relativistic" point of view is a complication of the axiomatic structure, a structure that can only be understood with the non-relativistic structure in mind.

The interpretation of $Totinf(\pi)$ makes it hard to attach a meaning to cases where ω, t are such that there is not a single pair ω', t' such that $Totinf(\pi)(\omega, t, \omega', t')$ is true. Nevertheless we do not formulate this as a general condition on $Totinf(\pi)$, partly since it still might turn out to come in handy for special kinds of abortion (cf. section 10)

On the basis of Totinf we can define a transitive relation between programs. If both π and σ are in $Prog(\Omega)$ we write

$$\pi \leqslant \sigma \qquad \text{iff} \qquad Totinf(\pi) \subset Totinf(\sigma)$$

If both $\pi \leqslant \sigma$ and $\sigma \leqslant \pi$ we say that π and σ are <u>semantically</u>

equivalent.

4.Semantics of primitive programs.

In this section we consider the programs "skip", "delay", "adlibitum", "nonterm" and a class of programs called "assignments". We ignore other primitive programs like x:= x + y, etc.

The programs "skip", "delay", "adlibitum" and "nonterm" will hardly ever occur in actual programs, but are just added to the collection of all programs in order to smooth the semantic discussion.

The program "skip" does nothing, that is to say that it leaves both ω and t untouched, at least as long as $t < \infty$. Its semantics is

$$Totinf(skip) = \lambda^* ((\omega = \omega') \wedge (t = t'))$$

The program "delay" is just like "skip" in as far as $\omega = \omega'$, but it "uses a unit of time":

$$Totinf(delay) = \lambda^* ((\omega = \omega') \wedge (t' = t + 1)).$$

The program "adlibitum" is a non-deterministic program that instructs the computer to do as it pleases, possibly even to use an infinite amount of time. (The term "adlibitum" is used in music with the same meaning, although one would not usually admit the performer to play infinitely long). The semantics is "no information at all", "anything may happen", and is expressed formally by

$$Totinf(adlibitum) = TRUE.$$

The program "nonterm" instructs the computer to go on for ever, and requires nothing about ω. Its semantics is

$$Totinf(nonterm) = \lambda(t' = \infty) = \lambda^*(t' = \infty) = NONTERM.$$

The trouble with a formal treatment of assignments is that

they contain expressions in some syntactic form, intended to represent elements of Ω, but the language in which they are formulated (the computer programming language) should not be confused with the (so much richer) mathematical language in which we discuss the semantics.

Let us say that somehow we have defined a class of expressions (in the programming language) and that to each E of that class we have assigned a mapping g of Ω into Ω. Then something like 'x := E' is a program; we note that E may contain a symbol x referring to the initial value ω, but no symbols referring to ω', t or t'. We shall not say precisely now g is obtained from E; one usually has the "naive" interpretation that the value $g(\omega)'$ is obtained if we just replace x by ω in E. But whatever the relation between E and g might be, the semantics is

$$\text{Totinf}(x:=E) = \lambda^* ((\omega' = g(\omega)) \wedge (t = t')).$$

We have chosen the (unrealistic) point of view that the execution of x:=E does not take time. If one wants to attach some time consumption to this program, it is easily administered by adding (in concatenation) a number of executions of 'delay'.

5. Lower primitive program constructs.

One of the simplest lower primitive program constructs is π or σ, if both π and σ are in $\text{Prog}(\Omega)$. The interpretation is that for every input the computer is free to choose which one of π and σ is to be executed. The semantics is described, if $P = \text{Totinf}(\pi)$, $Q = \text{Totinf}(\sigma)$, $R = \text{Totinf}(\pi \text{ or } \sigma)$, by

$$R(\omega, t, \omega', t') = P(\omega, t, \omega', t') \vee Q(\omega, t, \omega', t')$$

for all $(\omega, t, \omega', t') \in \Lambda$. Since both P and Q are constant at infinity, the same thing holds for R.

Next we consider the well-known construct 'π ; σ ', called the concatenation of the programs π and σ . The semantics is given by means of the convolution

$$Totinf(\pi ; \sigma) = Totinf(\pi) * Totinf(\sigma).$$

We note that $Totinf(\pi ; \sigma)$ is constant at infinity, since the second factor is constant at infinity. We also note that because of the associativity of the convolution the programs π ; (σ ; τ) and (π ; σ); τ are semantically equivalent.

With the construct 'if E then π else σ ' we have a situation similar to the one with the assignment statement we considered in section 4. Somehow we have defined a class of expressions in the programming language, and to each E of that class we have associated a predicate B on Ω . The expression E may contain a symbol x referring to the input value ω , but no symbols referring to ω ', t or t'.(The usual "naive" point of view is that the program text contains B itself, and that it reads "if B(x) then π else σ "). The intuitive (operational) meaning of "if E then π else σ " is that if $B(\omega)$ is true then π is to be executed, if $B(\omega)$ is false then σ is to be executed.

The formal semantics is described as follows. Let P, Q, R be the Totinf's of π , σ and "if E then π else σ ". Then we have

$$R(\omega, t, \omega', t') =$$

$$= (B(\omega) \wedge P(\omega, t, \omega', t')) \vee (\neg B(\omega) \wedge Q(\omega, t, \omega', t'))$$

and we note that the right-hand side is equivalent to

$$(B(\omega) \rightarrow P(\omega, t, \omega', t')) \wedge (\neg B(\omega) \rightarrow Q(\omega, t, \omega', t')).$$

We have to check that R is constant at infinity. This follows directly from the fact that both P and Q are constant at infinity.

In the lower primitive program constructs of this section

we have not administered any run time for the "overhead" of the constructs: the only time consumption is in the execution of the sub-programs π, σ, ... It would not be very hard to alter this by adding a number of executions of the program "delay".

6. Higher primitive program constructs

Let φ be a program-to-program function, i.e., a mapping of $Prog(\Omega)$ into itself. As an example we start from some expression E to which there corresponds a predicate B (like in section 4), and for any $\pi \in Prog(\Omega)$ we define $\varphi(\pi)$ by

$$\varphi(\pi) = \underline{\text{if}}\ E\ \underline{\text{then}}\ \pi\ \underline{\text{else}}\ \text{skip}.$$

For any program-to-program function φ we have a program to be called RECURS(φ). We may think of a program φ that can be described in ALGOL60 by the procedure declaration and procedure body

$$\underline{\text{procedure}}\ \varphi\ ;\ \hat{\varphi}(\varphi).$$

In order to describe the semantics of RECURS(φ) we first define the program-to-program function ψ by

$$\psi(\pi) = \varphi\ (\text{(delay; } \pi\text{)}). \qquad (1)$$

Next we introduce the functions ψ^k (k=0, 1, ...) by iteration: $\psi^0(\pi) = \pi$, $\psi^{k+1}(\pi) = \psi(\psi^k(\pi))$. We apply the ψ^k to the primitive program 'adlibitum'. Abbreviating

$$R_k = Totinf(\psi^k(adlibitum)) \qquad (2)$$

we now define the semantics of RECURS(φ) by

$$Totinf(RECURS(\varphi)) = R \qquad (3)$$

where

$$R(\omega, t, \omega', t') = \forall_{k \in \mathbb{N}} R_k(\omega, t, \omega', t') \qquad (4)$$

for all $\omega, \omega' \in \Omega$ and $t, t' \in T$. $\mathbb{N}$ is the set (1, 2, 3, ...), but it would do no harm to include k = 0 as well, since R_0 = Totinf(adlibitum) = TRUE.

We have to check that $R \in Pred^*(\Lambda)$, i.e., that R is constant at infinity. This is trivial from (4), since each R_k is constant at infinity.

In section 9 we shall comment on the definition (3), and discuss its relation to fixed point semantics. We postpone the discussion since we want to show first that (3) is good enough for a practical case: the "while"statement.

7. The while statement

We consider the program that is usually written as

$$\underline{\text{while}}\ E\ \underline{\text{do}}\ \tau, \qquad (1)$$

where τ is a program, and E is an expression that plays the same role as in section 4 in the construct "$\underline{\text{if}}\ E\ \underline{\text{then}}\ \pi\ \underline{\text{else}}\ \sigma$". We form a program-to-program function φ by

$$\varphi(\pi) = \underline{\text{if}}\ E\ \underline{\text{then}}\ (\tau;\pi)\ \underline{\text{else}}\ \text{skip} \qquad (2)$$

and we claim that the semantics of RECURS(φ) is able to explain the semantics usually attached to (1). We shall do this both for partial correctness and for total correctness. In both cases B is the predicate on Ω that corresponds to E.

Theorem 1 ("Partial correctness). Let C be a predicate on Ω and abbreviate

$$P = \lambda\ ((\ t=t') \wedge ((B(\omega) \wedge C(\omega)) \rightarrow C(\omega'))), \qquad (3)$$

$$F = \lambda((C(\omega) \wedge (t'<\infty)) \rightarrow ((C(\omega') \wedge \neg B(\omega'))). \qquad (4)$$

Assume that the semantics of τ satisfies

$$Totinf(\tau) \subset P^*. \qquad (5)$$

Then the one of RECURS(φ) satisfies

$$Totinf(RECURS(\varphi)) \subset F. \qquad (6)$$

Proof. As in section 6 we abbreviate

$$R_k = Totinf(\psi^k(\text{adlibitum})),$$

and we note that

$$\psi^{k+1}(\text{adlibitum}) =$$

$$= \underline{\text{if}}\ \Xi\ \underline{\text{then}}\ (\tau;\ \text{delay};\ \psi^{k}(\text{adlibitum}))\ \underline{\text{else}}\ \text{skip}.$$

By the rules of section 5 we get $R_{k+1} \subset W_{k+1}$ where

$$W_{k+1} = \lambda((\exists(\omega) \rightarrow H_k(\omega,t,\omega',t')) \wedge (\neg\exists(\omega) \rightarrow S(\omega,t,\omega',t'))), \quad (7)$$

$$H_k = \lambda(\exists_{\sigma s \sigma' s'} (D^*(\omega,t,\sigma,s) \wedge (s'=s+1) \wedge (\sigma'=\sigma) \wedge \wedge R_k(\sigma',s',\omega',T'))). \quad (8)$$

$$S = \text{Totinf}(\text{skip}) = \lambda^*((\omega=\omega') \wedge (t=t')),$$

By (3) we have

$$D^*(\omega,t,\omega',T') \rightarrow t=t'$$

and therefore (8) can be simplified to

$$H_k = \lambda(\exists_{\sigma\in\Omega} D^*(\omega,t,\sigma,t) \wedge R_k(\sigma,t+1,\omega',t')). \quad (9)$$

We define a sequence of predicates $P_k \in \text{Pred}(\Lambda)$ by

$$P_k = \lambda((C(\omega) \wedge (t'+1 \leq k+t) \wedge (t'<\infty)) \rightarrow \rightarrow (C(\omega') \wedge \neg\exists(\omega'))). \quad (10)$$

We remark that for all k

$$P_k = P_k^*. \quad (11)$$

i.e. that P_k is constant at infinity: we even have

$$P_k(\omega,t,\omega',\infty) \quad (12)$$

for all k,ω,t,ω' because of the subexpression $t'<\infty$ on the left in (10).

We shall show that

$$R_k \subset P_k \quad (k=0,1,2,\ldots). \quad (13)$$

Once this has been shown we rapidly get to (6): by section 6 $\text{Totinf}(\text{RECURS}(\varphi)) \subset R_k$ for all k, and for all $(\omega,t,\omega',t') \in \Lambda$ we have

$$(\forall_k P_k(\omega,t,\omega',t')) \rightarrow F(\omega,t,\omega',t') \quad (14)$$

If $t'=\infty$ this is trivial since the right-hand side is true, if $t'<\infty$ we can (given ω,t,ω',t') take k such that $t'+1 \leq k+t$, whence $P_k(\omega,t,\omega',t') \rightarrow F(\omega,t,\omega',t')$.

We shall prove (13) by induction. First

$$R_0 \subset P_0 \qquad (15)$$

is trivial: P = TRUE since $t'+1 \leq t$ is false on Λ. Next we take a fixed $k>0$, we assume $R_k \subset P_k$ and we shall show $R_{k+1} \subset P_{k+1}$. It suffices to show

$$W_{k+1} \subset P_{k+1}. \qquad (16)$$

We now fix $(\omega,t,\omega',t') \in \Lambda$, we assume

$$R_k \subset P_k \quad \text{and} \quad W_{k+1}(\omega,t,\omega',t') \qquad (17)$$

and our goal is $P_{k+1}(\omega,t,\omega',t')$. We split in two cases: $B(\omega)$ and $\neg B(\omega)$, so according to (15) and (7) we can reach our goal by proving

$$(H_k(\omega, t, \omega', t') \wedge B(\omega)) \rightarrow P_{k+1}(\omega, t, \omega', t'), \qquad (18)$$

$$(S(\omega, t, \omega', t') \wedge (\neg(B(\omega)))) \rightarrow P_{k+1}(\omega, t, \omega', t'), \qquad (19)$$

In order to show (18) we assume its left-hand side. By (9) and $R_k \subset P_k$ we conclude that σ exists such that

$$B(\omega) \wedge D^*(\omega, t, \sigma, t') \wedge P_k(\sigma, t+1, \omega', t'). \qquad (20)$$

If $t'<\infty$ we can just replace D^* by D, and from (20), (3), (4) we derive

$$(C(\omega) \wedge (t'+1 \leq k+t+1)) \rightarrow (C(\omega') \wedge \neg B(\omega')) \qquad (21)$$

and that means $P_{k+1}(\omega, t, \omega', t')$. If $t'=\infty$ we get $P_{k+1}(\omega, t, \omega', t')$ from (12).

Next we show (19). We assume $S(\omega, t, \omega', t')$, $\neg B(\omega)$, and have to prove $P_{k+1}(\omega, t, \omega', t')$. If $t'=\infty$ this is trivial by (12), so we take $t'<\infty$. We assume $C(\omega) \wedge (t'+1 \leq k+1+t) \wedge (t'<\infty)$ and want to show $C(\omega') \wedge \neg B(\omega')$. From $S(\omega, t, \omega', t')$ and $t'<\infty$ we get $\omega = \omega'$, so by $C(\omega)$ and $\neg B(\omega)$ we have $C(\omega') \wedge \neg B(\omega')$. This finishes the proof of Theorem 1.

Theorem 2 ("Total correctness"). Let C be a predicate on Ω, and let Q be a mapping of Ω into the set of integers ≥ 0. We abbreviate

$$D = \lambda(((B(\omega) \wedge C(\omega)) \rightarrow (C(\omega') \wedge (Q(\omega')<Q(\omega)))) \wedge (t=t')), \qquad (22)$$

$$F = \lambda((C(\omega) \wedge (t<\infty)) \rightarrow (C(\omega') \wedge (t' \leq t+Q(\omega)) \wedge \neg B(\omega'). \qquad (23)$$

Assume that the semantics of τ satisfies

$$\text{Totinf}(\tau) \subset D^*. \qquad (24)$$

Then the one of RECURS(φ) satisfies

$$\text{Totinf}(\text{RECURS}(\varphi)) \subset F \qquad (25)$$

Proof. With the new D and F we follow the proof of Theorem 1. We also take new P_k's:

$$P_k = \qquad (26)$$

$$= \lambda((C(\omega) \wedge (Q(\omega)<k) \wedge (t<\infty)) \rightarrow ((t'+1 \leq t+k) \wedge C(\omega') \wedge \neg B(\omega').$$

Since we have altered D, F and P_k, we have to supply new proofs for the details (11),(14),(15),(18),(19) of the proof of Theorem 1. We note that the simplification of (8) to (9) is again valid.

We have (11) since for $t' = \infty$

$$P_k(\omega, t, \omega', \infty) \leftrightarrow \neg(C(\omega) \wedge (Q(\omega)<k)) \vee (t=\infty)$$

and the right-hand side does not depend on ω'.

In order to show (14) we take any $(\omega, t, \omega', t') \in \Lambda$, we assume all $P_k(\omega, t, \omega', t')$ and $C(\omega) \wedge (t<\infty)$. Taking $k > Q(\omega)+1$ we infer from $P_k(\omega, t, \omega', t')$ that $t'+1 \leq t+k$ and $C(\omega') \wedge \neg B(\omega')$ and therefore $(t' \leq t+Q(\omega)) \wedge C(\omega') \wedge \neg B(\omega')$. Thus we have proved $F(\omega, t, \omega', t')$.

We now show (15): since $Q(\omega)<0$ is false for all ω we have from (26) P_0 = TRUE.

Next we take any $(\omega, t, \omega', t') \in \Lambda$ and any $k \in \{0,1,\ldots\}$ and we shall prove (18). We do this by showing that (20) leads to $P_{k+1}(\omega, t, \omega', t')$.

Assume (20). Moreover assume $C(\omega) \wedge (Q(\omega)<k+1) \wedge (t<\infty)$, and try to get $(t' \leq t+k) \wedge C(\omega') \wedge (\neg B(\omega'))$. The $D^*(\omega, t, \sigma, t)$ of (20) equals $D(\omega, t, \sigma, t)$ since $t<\infty$, and therefore implies in the present context $C(\sigma) \wedge ((Q(\sigma)<Q(\omega))$.

Hence $C(\sigma) \wedge (Q(\sigma) < k)$. The $P_k(\sigma, t+1, \omega', t')$ of (20) now gives $(t' \leq t+k) \wedge C(\omega') \wedge \neg B(\omega')$ as we wanted. This finishes the proof of (18).

We finally turn to (19). We assume (with some fixed $(\omega, t, \omega', t') \in \Lambda$ and fixed k) that $S(\omega, t, \omega', t') \wedge \neg B(\omega)$, and moreover $C(\omega) \wedge (Q(\omega) < k+1) \wedge (t < \infty)$. If $t' < \infty$ then $S(\omega, t, \omega', t')$ says that $\omega = \omega'$ and $t = t'$, and we get $(t'+1 \leq t+k+1) \leq C(\omega') \wedge \neg B(\omega')$. This proves $P_{k+1}(\omega, t, \omega', t')$. The case $t' = \infty$ does not occur, since $S(\omega, t, \omega', t')$ would still say $t = t'$, which conflicts with $t < \infty$. This finishes the proof of (19), and completes the proof of Theorem 1.

Sometimes we can definitely conclude to non-termination of a while-statement. Rather than stating general theorems we present a single example. The proof will again follow the pattern of the proof of Theorem 1.

In this example we take for Ω the set of all integers. The predicate B corresponding to the E in '<u>while</u> E <u>do</u> τ ' is given by $B(\omega) = (\omega > 0)$. And τ is a program for which we assume $Totinf(\tau) < D^*$, where

$$D = \lambda((t = t') \wedge (\omega' = \omega + 1)). \qquad (27)$$

(in ALGOL one can think of the while-statement '<u>while</u> x > 0 <u>do</u> x: = x + 1'). We shall derive the semantical statement $Totinf(RECURS(\varphi)) < F$, where

$$F = \lambda((\omega > 0) \rightarrow (t' = \infty)). \qquad (28)$$

This corresponds to the intuitively obvious statement that if the initial value of x is positive, then '<u>while</u> x > 0 <u>do</u> x: = x + 1' is non-terminating.

Again we follow the proof of Theorem 1. We take

$$P_k = \lambda((\omega > 0) \rightarrow (t + k \leq t' \leq \infty)). \qquad (29)$$

Since $D^*(\omega, t, \omega', t') \rightarrow (t = t')$ we can again simplify (8) to (9).

Now again it suffices to check (11), (14), (15), (18) and (19).

We have (11) since ω' does not occur in (29). And (14) is trivial by (29) and (28). Next (15) is obvious since $P_0 =$ TRUE (note that $t \leq t' \leq \infty$ in all points of Λ).

Now we turn to (18). We assume the left-hand side, i.e. $H_k(\omega,t,\omega',t')$ and $\omega > 0$. Turning to (9) we note that $D^*(\omega,t,\sigma,t)$ implies $(t=\infty) \wedge (\sigma=\omega+1)$. So by the induction assumption $R_k \subset P_k$ we derive from $H_k(\omega,t,\omega',t')$ that σ exists such that

$$((t=\infty) \vee (\sigma=\omega+1)) \wedge ((\sigma > 0) \rightarrow (t+1+k \leq t' \leq \infty)).$$

If $\omega > 0$ we deduce $(t=\infty) \vee (\sigma > 0)$, so $(t=\infty) \vee (t+1+k \leq t' \leq \infty)$, whence $t+1+k \leq t' \leq \infty$.

Therefore we have proved (18).

Finally (19) is trivial since already $\neg B(\omega)$ implies $P_{k+1}(\omega, t, \omega', t')$ by (29).

8. A contemplation on syntax and semantics

There is a world of semantics and a world of syntax. We use the word "world" in order to avoid to have to be very precise. It means something like "area of attention". Let us call these worlds SEM and SYN.

In SEM we talk about sets, relations and mappings in the usual mathematical sense. These mathematical "objects" are discussed in ordinary mathematical language.

In SYN we talk about strings of characters, and in particular about special strings which are called "programs". Again we use ordinary mathematical language to discuss these linguistic objects.

So both for SEM and for SYN we use mathematical language, but the "objects" are different. Gradually we discover possibilities to link objects in SYN to objects in SEM, but there

is always trouble with the metalanguages of SEM and SYN, in particular in those cases where we use one and the same word (like "variable") in different meanings in the two metalanguages.

In spite of the formidable amount of knowledge about formal languages it must be said that SYN is a poor man's world, an underdeveloped country. SYN cannot really live without SEM, but SEM can certainly live very comfortably without SYN, just by developing a bit of extra metalanguage.

Let us compare the situation of computer programs with a subject that came up about two thousand years earlier: geometrical constructions with ruler and compass.

In this geometrical case SEM is the world of geometrical objects and logical discussions about those objects. Since the whole of mathematics is available to SEM, it includes sets and mappings. As an example we mention that there is a mapping that attaches to each pair (P, r), where P is a point and r a line segment, the set circle(P,r), which is the set of all points in our plane which have the distance r to P. Let us call the act of getting the set circle(P, r) from P and r a construction.

In the metalanguage we now describe sequences of such constructs, which lead from a set of objects we are assumed to "have" at the start, to the objects which somehow interest us. We say that this sequence constructs these interesting objects. Parallel to this sequence we have a sequence of actions in our physical world on physical paper with physical ruler, compass and pencil, and for this sequence of physical actions the sequence in SEM is a mathematical model. But we have to emphasize that the world of SEM is bigger that this. We might study constructions for which no physical realization is available.

Now where is SYN in this case? Let us hire people to carry out the geometrical constructions we invented. Assume these people are unable to understand our metalanguage. We have to instruct them very precisely what to do at each step. To that end we invent a system for coding instructions, and these coded instructions are the programs of SYN. (One might say tha LOGO is a kind of programming language for at least some geometrical constructions.)

The question of whether a sequence of commands in SYN corresponds exactly to the sequence we had in in SEM's metalanguage, is independent of whether we actually execute or can execute these commands physically.

Let us now get to computer programs. The historical order seems to be somewhat different from the old geometrical case. Most of it started with programs (which had to be very precisely defined) plus a somewhat informal notion of state space and a possibly even more informal notion of time. At the moment the need for "program correctness proofs" was felt, SYN was much better developed than SEM. It is quite natural that this resulted in various ways to treat program correctness which were mostly SYN-centered. It became SYN with a tiny bit of SEM (like state space and predicates), or SYN with a lot of SEM (like fixed point theory). Even the term "program correctness " itself bears the traces of this. The term suggests syntactical correctness, but means something different: it means that a program is correct with respect to some semantical specification.

In the geometrical case one of course feels that the matter of correctness of a geometrical construction (like the question of whether our construction for a regular pentagon really leads to a pentagon that is regular) is a matter of SEM only, and that the question has nothing to do with the way we have coded the construction in SEM. Yet we can of course raise the question

whether the execution of a given coded description of a construction leads to a proper pentagon. But it would be a clear case for "separation of concerns" to split this question into (i) whether the program is correct, and (ii) whether the coding is correct.

There does not seem to be a good reason for tying things to SYN. In ordinary mathematical language we can define anything we need, like sets and mappings, without ever bothering about the kind of notation we use. There is a strong notion of equality in mathematics, with the effect that one and the same object can be described in various syntactically different ways. This is true for "objects" as well as for "actions".

So let us try to keep the matter of program semantics away from SYN. In SEM we can express in ordinary mathematical language everything we want for program correctness, and in an automatic checking system (like AUTOMATH) we can speak of integrated semantics. In integrated semantics we can describe logic, mathematics, programs and program semantics all in one and the same system. And a compiler would be able to read these mathematically defined programs and to translate them into machine language without ever passing the computer languages we usually think of.

The total effect of integrated semantics will be simplification. It might also be a satisfactory framework in which other semantical systems can be placed and compared.

In SYN-free semantics one can consider programs which are not representally syntactically at all. It might be possible to characterize the representable programs in the set of all programs by means of properties like monotonicity (see section 8), and show that things like fixed-point theory can be developed on the basis of such properties. But why go into all that trouble? In section 6 we showed an example where

an important ingredient of practical programming was treated semantically without any reference to such properties, and it seems likely that we can go quite a distance in this style.

IN SYN-free semantics it seems to be attractive to identify the notions of a "program" with the notion of the semantic information of that program. Yet there is something to say for the idea of creating a separate set (the set of programs) and to map it into the set of relations by a mapping "Totinf", as we did in section 2. This policy anyway leaves various possibilities open. In particular we keep the possibility to add equality and equivalence assumptions in the set of programs, and such assumptions might be adjustable to later mappings of the programs in SYN into this set of programs.

So we do not require that every element of Prog(Ω) is representable in SYN, and we do not require that two elements of Prog(Ω) are equal if they have the same semantics.

Note that sometimes the notion of equality in SYN might be stronger that the one in Prog(Ω). For example, the repeated concatenations

x:=x+1; (x:=x+2; x:=x-1)

(x:=x+1; x:=x+2); x:=x-1

might be considered as equal in SYN, and their counterparts in Prog(Ω) are different but semantically equivalent. On the other hand the programs

x:=x+1; x:=x+3

x:=x+2; x:=x+2

will be considered to be different in SYN, as well as different in Prog(Ω), but yet semantically equivalent.

For the time being we just leave it open what Prog(Ω) is. Followers of SYN-ful semantics might like to identify it with the set of all their programs, and their antagonists might like to

identify it with the set of all relations, like Pred(Λ).

9. Comments on the semantics of recursion.

In section 6 we defined the semantics of RECURS (for any program-to-program function φ) by means of formulas (1), (2), (3), (4). In this section we give some arguments for this choice, and we compare it with other possibilities.

In the process of evaluating and comparing we shall appeal to more or less intuitive ideas on the structure and execution of computer programs. It should be stipulated that those ideas are certainly not substantiated in all respects by the treatment of program semantics as explained thus far in this paper.

Let us first ignore the ψ of section 6, and just work with the iterates of φ itself. If μ is a program then

$$\mu,\ \varphi(\mu),\ \varphi^2(\mu),\ \varphi^3(\mu),\ \ldots$$

are programs. In order to facilitate the discussion we assume for a moment that all these programs are deterministic. We take any initial value ω, and we ask what happens in the execution. Our intuition says: either the recursive program is non-terminating, or there is a k such that in the execution of $\varphi^k(\mu)$ the μ is not executed at all. This also means that the executions of $\varphi^k(\mu)$, $\varphi^{k+1}(\mu)$, ... are all equal, as far as the initial value ω is concerned, and that they are all equal to the executions of $\varphi^k(\nu)$, $\varphi^{k+1}(\nu)$, ... for any other program ν. In the case of non-deterministic programs these things are harder to explain, but the idea remains the same.

In the preparation of note [1] this idea led to a particular choice of μ : μ is a program with

$$\text{Totinf}(\mu) = \lambda(t' = \infty),$$

which means that every execution of μ is non-terminating. Consequently: if any execution of $\varphi^k(\mu)$ actually executes μ,

then that execution of $\varphi^k(\mu)$ is non-terminating too. So we get the semantics we expect, if we say that (ω, t, ω', t') satisfies the predicate $Totinf(\varphi^k(\mu))$ for all large k (or at least for infinitely many k). In the case of termination the μ in $\varphi^k(\mu)$ is not executed at all if k is large. In the case of non-termination the μ is executed in all these $\varphi^k(\mu)$, and that takes care of the truth of $Totinf(\varphi^k(\mu))$ at $(\omega, t, \omega', \infty)$ for some ω'.

The objection one might make is the lack of monotonicity in the sequence. Let us discuss monotonicity first in general terms. If φ is a program-to-program function we can "expect" φ to be monotonic in the sense that

$$\pi \leq \sigma \quad \rightarrow \quad \varphi(\pi) \leq \varphi(\sigma)$$

(with $\leq$ defined as in section 3) for all programs π, σ: if we know more about π than about σ, then we know more about $\varphi(\pi)$ than about $\varphi(\sigma)$. Unfortunately we cannot apply this in the case of the sequence $\mu, \varphi(\mu), \varphi^2(\mu), \ldots$ since there is no guarantee that either $\mu \leq \varphi(\mu)$ or $\varphi(\mu) \leq \mu$ for the "non-termination" program μ. We are so much better off with adlibitum: $adlibitum \geq \varphi(adlibitum)$, and if φ is monotonic this leads to

$$adlibitum \geq \varphi(adlibitum) \geq \varphi^2(adlibitum) \geq \ldots$$

Unfortunately we do not get the proper semantics this way. The simplest example is the one where φ is the identity: $\varphi(\pi) = \pi$ for all programs π. Then the limit of the sequence with entries $Totinf(\varphi^k(adlibitum))$ is just $Totinf(adlibitum)$. This means that the sequence provides no semantic information at all: it just says that anything may happen, and not that RECURS(φ) is definitely non-terminating.

This objection has been overcome in section 6 of this paper by taking ψ instead of φ. The extra executions of "delay" have the effect that for this particular φ

$$Totinf(\psi^k(adlibitum)) = \lambda(t+k \leq t' \leq \infty).$$

Taking limits for $k \to \infty$ we get $\lambda(t'=\infty)$, and that is what we wanted. Here we use the following definition of the limit of a monotonically decreasing sequence

$$\pi_1 \geq \pi_2 \geq \pi_3 \geq \quad \ldots \qquad (1)$$

We say that

$$\pi_k \to \pi \qquad (k \to \infty) \qquad (2)$$

if for all (ω, t, ω', t')

$$\forall_k \; P_k(\omega, t, \omega', t')) = P(\omega, t, \omega', t'),$$

where P_k =Totinf(π_k), P = Totinf(π). We note that Totinf(π) is uniquely determined by the sequence $\pi_1, \pi_2, \ldots$, and moreover that

$$\pi_k \geq \pi \qquad (3)$$

for all k.

In [1] we did not have monotonicity of the sequence of programs, and we had to have "lim sup" instead of "lim". The fact that we have monotonicity in the present semantics has the obvious advantage that the lim of a monotonic sequence is nicer to deal with than the lim sup of a non-monotonic sequence. If we have (1) and (2), then for any arbitrary k we can use Totinf(π_k) as information about π, since (3) expresses Totinf(π_k) $\geq$ Totinf(π). This is much simpler than with lim sup, where we can obtain information about the lim sup only if we have information about Totinf(π_k) for infinitely many k.

Let us now discuss the idea of RECURS(φ) being a fixed point. Intuitive ideas about execution suggest that the "fixed point statement"

Totinf(φRECURS(φ)) = Totinf(RECURS(φ)), (4)

but it is not easy to actually prove this without making restrictive assumptions about the class of constructs we take φ from. Just monotonicity will not do. Monotonicity does suffice for the weaker result

$$\psi(\text{RECURS}(\psi)) \leq \text{RECURS}(\psi). \qquad (5)$$

This follows if we apply ψ to both sides of the inequality (cf.(3))

$$\text{RECURS}(\psi) \leq \psi^{k}(\text{adlibitum})$$

and take limits for $k \to \infty$.

The equality (4) is easy if we assume that ψ is not just monotonic but also continuous. We take the latter notion in the sense that

$$\lim \psi(\pi_k) = \psi(\lim(\pi_k)) \qquad (6)$$

for any sequence satisfying (1).

However, it may be quite hard to establish continuity, even in this weak sense, for all ψ's arising from a given set of program constructs. In particular we have to bear in mind that that we may wish to apply RECURS to functions ψ which in their definition contain applications RECURS already.

If we take our set of program constructs a bit too wide, it is easy to kill continuity even for very simple program functions. We shall present an example that might give an idea of the kind of restrictions we might have to build in.

Let Ω be the set of integers, and imagine that for every k we have a program ~~with~~ π_k with $Totinf(\pi_k) = \lambda^{*}((\omega' \geq k) \wedge (t=t')) \vee (t'=\infty))$. Let σ be described similarly by

$$Totinf(\sigma) = \lambda^{*}((\omega'=0) \wedge (t=t')).$$

We define the program-to-program function θ by means of concatenation with τ :

$$\theta(\tau) = (\tau ; \sigma)$$

for all τ.

We have $\pi_1 \geq \pi_2 \geq \ldots$ .... If we call the limit π it follows from the monotonicity of ψ that $\psi(\pi_1) \geq \psi(\pi_2) \geq \ldots$ Let us put $\rho = \lim \psi(\pi_k)$. We hope that ρ and $\theta(\pi)$ are semantically equivalent, but unfortunately this is not the case:

$$Totinf(\rho) = \lambda^{*}(((\omega'=0) \wedge (t=t')) \vee (t'=\infty))$$

$$Totinf(\theta(\pi)) = \lambda^{*}(t'=\infty).$$

This means that a general proof of (4), which means (6) with $\pi_k = \psi^k$(adlibitum) has to depend on more knowledge about the sequence π_k then just monotonicity.

Another point that has to be raised is the maximality. Let us assume that F is a "predicate transformer" which is such that Totinf($\psi(\pi)$) = F(Totinf(π)) for all π. Then if P = = Totinf(RECURS(ψ)) we can read (4) as

$$F(P) = P.$$

Now let Q be any other predicate with $F(Q) = Q$. Then just assuming monotonicity we can show that $Q \subset P$, in other words: P is the maximal fixed point of F. In order to show $Q \subset P$ we remark that

$$TRUE \supset F(TRUE) \supset F^2(TRUE), \quad \ldots$$

and that the limit of the sequence $F^k(TRUE)$ is P. Comparing this with the sequence

$$Q, \quad F(Q), \quad F^2(Q), \ \ldots$$

(of which all entries equal Q) we infer from $Q \subset TRUE$, by monotonicity of F, that $Q \subset F^k(TRUE)$, and therefore $Q \subset P$.

The question arises whether it is really worth the trouble of finding satisfactory restrictions on ψ that guarantee (4). After all, we have shown in section 7 that we can get to quite practical statements on actual programs without ever going into notions like continuity and fixed points. We did not even have to mention monotonicity!

It might be easier to prove (4) under restrictive assumptions like finiteness of state space, or exclusion of non-determinism. But such restrictions do not seem to be attractive for the practical discussion of actual programs.

Anyway there is quite a distance between the definition of recursion semantics by means of lim(Totinf (ψ^k(adlibidum))) and any definition based on the idea of a maximal fixed point.

It is a matter of opinion which one of the two ideas is preferable as the definition of the semantics of recursion.

10. A comment on abortion

We have not yet described the notion of abortion in our semantical system.

Here we first discuss an attempt to treat abortion in a way that seems to be promising at first, and we shall explain why it is not satisfactory.

The attempt is this one. If π is a program, and ω, t are initial state and time such that there do not exist any ω', t' (not even with $t'=\infty$) such that Totinf(ω, t, ω', t') holds, then we might try to interpret this as abortion. This means: with the initial ω, t the execution of π will have been interrupted at some point. The semantical system does not disclose at exactly which point further execution is refused by the machine, simply because it never discusses executional details.

This point of view seems to be very promising for Dijkstra's guarded command statement

$$\underline{\text{if}}\ E_1 \to \pi_1 \ \square\ \ldots\ \square\ E_k \to \pi_k\ \ \underline{\text{fi}} \qquad (1)$$

where each E_i corresponds to a predicate B_i (like in our discussion of the "<u>if</u> <u>then</u> <u>else</u>" in section 4). The semantics is: "select at random an i such that $B_i(\omega)$ holds, and then execute π_i ; if there is no such i then abort". If abortion is interpreted in the style "no possible ω', t'" we indicated above, then the total information of the program (1) is given by

$$(B_1(\omega) \wedge P_1(\omega,t,\omega',t')) \vee \ldots \vee (B_k\omega) \wedge P_k(\omega,t,\omega',t')), \qquad (2)$$

where P_i = Totinf(π_i) (i=1,...,k). The simplicity of (2) seems to be a positive point both for Dijkstra's semantics and

for the "no possible ω',t'" interpretation of abortion.

Unfortunately we have to admit that the "no possible ω',t'" interpretation of abortion conflicts with the idea of nondeterministic programs. We show this with a concatenation "$\pi;\sigma$", where π is a nondeterministic program and σ is a program that sometimes aborts. Let b and c be two different elements of Ω, and let

$$Totinf(\pi)(\omega, t, \omega', t') = ((\omega'=b) \vee (\omega'=c)) \wedge (t=t')$$

$$Totinf(\pi)(\omega, t, \omega', t') = (\omega \neq b) \wedge (\omega=\omega') \wedge (t=t').$$

So σ leads to abortion with the initial state b, but is harmless with all other initial states. By our semantic rule on concatenation we have

$$Totinf(\pi;\sigma)(\omega, t, \omega', t') = (\omega'=c) \wedge (t'=t) \qquad (3)$$

so here there is no abortion for any ω,t. Therefore (3) does not describe the situation adequately, since we expect that the semantics of "$\pi;\sigma$" is: whatever ω,t is, we have either abortion or $(\omega'=c) \wedge (t'=t)$.

A more satisfactory way to incorporate abortion into a semantic system is by means of a special boolean variable. Let us call it ab (for "abortion"). At the start of a program we add the assignment "ab:=false" (interpretation: no abortion thus far), and every sub-program σ of the program is replaced by "if $\neg$ ab then σ else skip". If in the final state we have ab = true then we interpret this by saying that the program execution has been aborted. In addition to this the sub-programs may contain assignments "ab:=true" in those cases where we actually want abortion to take place. We might want to do this in the guarded command statement. Another example is overflow: if we do not wish to handle numbers exceeding 1/m, we might transform "y:=1/p" into "if |1/p| $\geqslant$ m then y: = 1/p else ab: = true".

Let us use the word "refuser" for such an extra boolean variable like ab. The essential thing for a refuser r is that subprograms σ are to be remodelled into "if $\neg$ r then σ else skip". Refusers can be used for semantic discussion of forward goto's as well.

Properly speaking, if we treat abortion with refusers, we seem to need a refuser for the indication of nontermination too. We illustrate this by a concatenation "$\pi ; \sigma$", where π is a program that terminates for no input, and σ aborts for every input. The verdict for "$\pi ; \sigma$" should be that it is non-terminating, and not that it is non-terminating as well as aborting. So again the semantic discussion should be as if σ is skipped in cases where π is non-terminating.

REFERENCE

1. N. G. de Bruijn,
A system for handling syntax and semantics of computer programs in terms if the mathematical language AUTOMATH. Report, Department of mathematics, Technological University Eindhoven, The Netherlands. November 1973.

100000 The two kinds of semantics get closer together if we
100150 take a more liberal interpretation of non-termination. In this
100200 more liberal version a semantical statement that says (with
100250 ω,t given), that t' = ∞ is a possible effect, has to be inter-
100300 preted as that there is no upper bound to the values of the t'
100350 of the possible executions (with initial values ω,t). After
100400 all, if we are interested in having our programs terminated,
100450 a statement that a program might run for a million years does
100500 not give us much more comfort than a statement that it might
100550 go on for ever. Therefore it is quite reasonable to identify
100600 "unpredictably long" with "infinitely long".
100650
100700 This "liberal version" is related to the following defi-
100750 nition. If $P \in Pred(\Lambda)$ then instead of the P^* of section 2
100800 we define P^+ by: $P^+(\omega,t,\omega',t') = P(\omega,t,\omega',t')$ if
100850 $t' < \infty$, and
100900
100950 $$P^+(\omega,t,\omega',\infty) =$$
101000 $$= \forall_{u \in T\setminus\{\infty\}} \exists_{v \in T | v > u} \exists_{\rho \in \Omega} P(\omega,t,\rho,v).$$
101050
101100 If the set of all P^+ with $P \in Pred(\Lambda)$ is denoted by $Pred^+(\Lambda)$,
101150 we have
101200
101250 $$Pred^+(\Lambda) \subset Pred^*(\Lambda) \subset Pred(\Lambda).$$
101300 In deterministic cases there is no difference between P^*
101350 and P^+. To be more precise, if ω,t are such that there is at
101400 most one pair ω',t' with $t' < \infty$ and $P(\omega,t,\omega',t')$, then
101450 $P^+(\omega,t,\omega',t') = P^*(\omega,t,\omega',t')$ for all ω',t'.
101500
101550 In general, if P describes the semantics of a program
101600 in the original version (where $P(\omega,t,\omega',\infty)$ means that the
101650 program can actually run for ever), then P^+ is the liberal
101700 semantics (where $P^+(\omega,t,\omega',\infty)$ means that there is no

103050
103100 upper bound to the runtime).
103150
103200 If we take the liberal semantics, then our prospects for
103250
103300 proving the fixed point property for recursion become much more
103350
103400 promising.
103450
103500
103550
103600 The difference between P^* and $P^\dagger$, and its being related
103650
103700 to having non-determinism and infinite state space, can be
103750
103800 connected to König's well known infinity lemma. In order to
103850
103900 make the notation sufficiently clear for further discussion,
103950
104000 we explain it in a few words.
104050
104100 Let (S,r,f) be a rooted tree. That is, S is a set
104150
104200 (the set of "points"), r is a special element of S
104250
104300 (called the "root") and f is a mapping of $S\backslash\{r\}$ into S
104350
104400 (f(x) is called the "father" of x). It is assumed that for
104450
104500 every x ($x \neq r$) there is an integer n such that the n-th
104550
104600 iterate f^n maps x into r. This n is uniquely determined,
104650
104700 and is called the "level" of x. The level of r is zero. If
104750
104800 $x \in S$, the set of all $y \in S\backslash\{r\}$ with f(y)=x is called the
104850
104900 "offspring" of x, and denoted O(x).
104950
105000 If $x \in S$ we denote by IP(x) the proposition that there
105050
105100 is an infinite path starting from x, that is a sequence
105150
105200 $x_0, x_1, x_2, \ldots$ with $x_0 = x$ and $f(x_{n+1}) = x_n$ for $n = 0,1,2,\ldots$.
105300
105350
105400 And by UL(x) (UL abbreviates "unbounded level") we denote
105450
105500 the proposition that for every natural number m there is a path
105550
105600 $x_0, \ldots, x_m$, again with $x_0 = x$, $f(x_{n+1}) = x_n$ for $n = 0,\ldots,m-1$.
105700
105750
105800 We note that for all x
105850
105900 $$IP(x) \rightarrow UL(x).$$
105950
106050
106100 Finally we formulate the "finite offspring condition".
106150
106200 It says that for every $x \in S$ the offspring O(x) is a finite
106250

set.

Konig's lemma expresses: If the finite offspring condition holds, and UL(r) is true, then we have IP(r).

Coming back to semantics, we shall try to explain that IP(r) can be compared with infinite runtime, and UP(r) with unpredictably long runtime, in both cases with initial value r. And the finite offspring condition corresponds to a condition that says that in the number of

possible outputs is finite, for every given input. This condition is certainly guaranteed if the state space is finite, but also if the program is deterministic.

We describe a typical case of a tree where UL(r) holds but IP(r) does not. Let us call it (S ,r ,f). The points of S are the pairs (i,j) with integers i,j satisfying either $0 < i \leq j$ or $i = j = 0$. The point (0,0) is taken as the root. For all other points we define the "father" by

$$f_0(i,j) = \begin{cases} (i-1,j) & \text{if } (0 < i \leq j) \wedge (i \neq 1) \\ (0,0) & \text{if } 1 = i \leq j. \end{cases}$$

The tree is depicted in figure 1. In that figure the arrows run from points to their fathers.

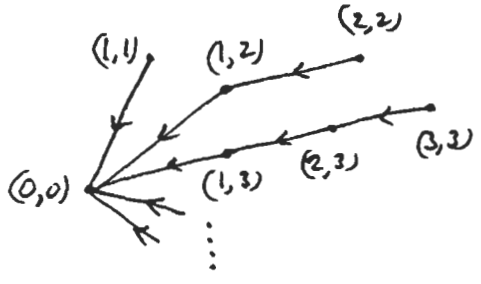

Figure 1

Coming back to the general tree (S, r, f), we describe programs for which S is the state space. As a primitive program we take the program "step". Its semantics is described by

$$Totinf(step) = (t'=t) \wedge (\omega' \in O(\omega)).$$

Note that "step" is a non-deterministic program, at least if there exist points x where O(x) has more than one element. If $O(\omega)$ is empty (such an x is called an end-point), then there is no possible output ω' to the input ω. If this is considered unacceptable, one might take any arbitrary value of ω' as output, like $\omega' = \omega$. That means that in the definition of Totinf(step) we replace $\omega' \in O(\omega)$ by

$$\omega' \in O(\omega) \vee ((O(\omega) = \phi) \wedge (\omega' = \omega)).$$

But actually the case of empty $O(\omega)$ is unimportant since the program "step" will not be used there.

Let us now discuss the program (see the beginning of section 7)

$$\underline{\text{while}}\ E\ \underline{\text{do}}\ step,$$

where E corresponds to the predicate B given by

$$B(\omega) = (O(\omega) \neq \phi).$$

In the notation of formula (2) of section 7 this program denotes RECURS(φ), where φ is given by

$$\varphi(\pi) = \underline{\text{if}}\ E\ \underline{\text{then}}\ (step\ ;\pi)\ \underline{\text{else}}\ skip.$$

The "intuitive", or, if one prefers, "operational" semantics of this program is the following one. Let ω (the input) be any point of the tree. If ω' is an end-point such that $f^k(\omega') = \omega$ for some $k \geq 0$ then ω' is a possible output (and $t'=t+k$). If $IP(\omega)$ holds then there is a non-terminating execution. If $IP(\omega)$ is false but $UL(\omega)$ still holds then there exist unpredictably long executions. If $UL(\omega)$ does not hold then all executions starting with ω terminate, and there is an upper bound to their runtime.

Let us now investigate Totinf(RECURS(ω)) as defined by (4) in section 6. As far as terminating executions are concerned, it produces the same results as the intuitive

112750
112800 semantics. For values of ω where IP(ω) holds, it proclaims
112850
112900 the possibility that t'= ∞, as it should. But in points where
112950
113000 UL(ω) holds but IP(ω) does not, the semantics of section 6 still
113050
113100 says that t' = ∞ is possible, and the intuitive semantics says it
113150
113200 is not.
113250
113300 The difference between these two kinds of semantics vanishes
113350
113400 (at least for this program) if we identify "unpredictably long
113450
113500 runtime" and "non-termination".
113550
113600 In the tree program described here, it is also easy to
113650
113700 illustrate that ψ(RECURS(φ)) and RECURS(φ) can have different
113750
113800 semantics in the system of section 6. We take
113850
113900 ~~semantics. We take~~ the special tree (S_o, r_o, f_o). We have UL(r_o),
113950
114000 but UL(x) is false for all x with f(x) = r_o. From this it can be
114050
114100 derived that at (r_o, t, ω', ∞) (where ω' irrelevant),
114150
114200 RECURS(φ) is true but ψ(RECURS(φ)) is not.
114250 Things look much better in the more liberal Pred$^+$ semantics. We briefly discuss the changes that have to be made.
114300
114350 First, in section 2 we have to introduce Pred$^+$ (Λ) instead
114400
114450 of Pred* (Λ). We have to give up the idea of a J such that always
114500
114550 P $*$ J = P*, like in formula (2) of section 2, and it is not
114600
114650 generally true that (P $*$ Q)$^+$ = P $*$ Q$^+$ (A counterexample:
114700
114750 Ω= $\mathbb{N}$, P(ω, t, ω', t') = (t' = t), Q(ω, t, ω', t') =
114800
114850 = ((ω' = ω) $\wedge$ t' = t + ω)).
114900
114950 ~~In section 5 the semantics of the concatenation has to be des-~~
115000
115050 ~~described by~~
115100
115150 ~~Totinf(;) (Totinf( ) Totinf( )).~~
115200
115250 In section 3 we have to introduce Totinf$^+$ as a mapping
115300
115350 of Prog(Ω) into Pred$^+$ (Λ). The changes in section 4 are
115400
115450 trivial.
115500
115550 In section 5 the semantics of the concatenation has to be
115600
115650 described by
115700
115750 Totinf(π ;σ) = (Totinf(π) $*$ Totinf(σ))$^+$.
115800

In section 6 the semantics of RECURS(φ) can be given by

$$\text{Totinf}\,(\text{RECURS}(\varphi)) = R^{+},$$

although the definition of R in (4) will guarantee that there is no difference between R and R^{+} as soon as we have monotonicity. We note that if $P_0 \supset P_1 \supset P_2 \supset \ldots$ with $P_n = P_n^{+}$ for all n, and if $P = \bigwedge_n P_n$, then $P = P^{+}$.

In Theorems 1 and 2 of section 7 the new semantics makes no difference, since they deal with cases with bounded runtime. In the non-termination example at the end of section 7 there is neither any difference, since the program is deterministic.

From the tree-program discussed earlier in this section it can be seen that ψ(RECURS(φ)) and RECURS(φ) need not have the same Pred^{*}-semantics in nondeterministic cases with infinite state space. If we turn to Pred^{+}-semantics, however, it seems that we only need monotonicity properties in order to show that

$$\text{Totinf}^{+}(\psi\text{RECURS}(\varphi)) = \text{Totinf}^{+}(\text{RECURS}(\varphi)),$$

which means that RECURS(φ) can be considered as the maximal fixed point.