

100000
100050
100100
100150
100200
100250
100300
100350
100400
100450
100500
100550
100600
100650
100700
100750
100800
100850
100900
100950
101000
101050
101100
101150
101200
101250
101300
101350
101400
101450
101500
101550
101600
101650
101700
101750
101800
101850
101900
101950
102000
102050
102100
102150
102200
102250
102300
102350
102400
102450
102500
102550
102600
102650
102700
102750
102800
102850

COMPUTER PROGRAM SEMANTICS IN SPACE AND TIME

by N. B. de Bruijn

1. INTRODUCTION.

This note can be considered as an addition to a ten year old note [1]. We give a new treatment of recursive procedures, i.e. a new version of section 11 of [1]. This new treatment also affects 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 AUTOMAT¹, had to introduce a new notion of order between predicates and had to impose slightly awkward monotonicity restrictions in order to establish a correspondence between the

102900 recursion semantics of LL and fixed point theory. The system proposed
 102950 in this note is much easier in this respect. Yet, if we weaken the
 103000 runtime information by distinguishing between finite and infinite
 103050 runtime only, the semantics can be expected to be the one of LL.

103100 Our present system takes care of runtime by describing a relation
 103150 between the moment t where the execution of a program starts, and the
 103200 moment t' where the execution ends. Essential information
 103250 about a program will have the form of a predicate on $(\Omega \times T) \times$
 103300 $\times (\Omega \times T)$ (whereas in LL it had the form of a predicate on
 103350 $\Omega^+ \times \Omega^+$. It is quite reasonable to think of predicates in which

103400 the relation between t and t' is expressed in a form like
 103450 $t + p(\omega, \omega') \leq t' \leq t + q(\omega, \omega')$,

104000 and it is also quite reasonable to choose the semantics of primitive
 104050 program constructs in accordance with this form. Nevertheless we
 104100 shall not set it as a rule that our predicates should necessarily
 104150 have this form. What we do require, however, is that $t \leq t'$ is somehow
 104200 enforced, and we shall realize this by restricting our predicates to
 104250 the set of all those quadruples (ω, t, ω', t') for which $t \leq t'$.

104300 The interpretation of t and t' is obvious. In a case where
 104350 $t < \infty$, $t' = \infty$ we of course say that the program does not terminate.
 104400 In cases where $t = \infty$, $t' = \infty$ we can say that the program execution
 104450 never started, since some other program had to be executed first,
 104500 took infinitely long.

104550 In order to be able to say that the sum of the lengths of
 104600 infinitely many time intervals is infinite, we restrict ourselves
 104650 to time moments which are either integers or the symbol ∞ .

104700 Just like in LL, where programs never explicitly referred
 104750 to the element ∞ , in our present system the programs will usually
 104800 not refer to t in any way, except for the program "delay".

104850 In this note we shall discuss the relation between t and t'
 104900 only as far as it is relevant for the treatment of recursion. In
 104950

106000
106050
106100
106150
106200
106250
106300
106350
106400
106450
106500
106550
106600
106650
106700
106750
106800
106850
106900
106950
107000
107050
107100
107150
107200
107250
107300
107350
107400
107450
107500
107550
107600
107650
107700
107750
107800
107850
107900
107950
108000
108050
108100
108150
108200
108250
108300
108350
108400
108450
108500
108550
108600
108650
108700
108750
108800
108850
108900
108950
109000
109050

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 $\text{Pred}(\Lambda)$ 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

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

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 sense to distinguish between different initial pairs (ω, ∞) , but there it causes no trouble for our semantic system). So now 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 $P \in \text{Pred}(\Lambda)$ is constant at infinity if $P(\omega, t, \omega_1', \infty) = P(\omega, t, \omega_2', \infty)$ for all $\omega, \omega_1', \omega_2' \in \Omega$ and all $t \in T$. The set of all $P \in \text{Pred}(\Lambda)$ which are constant at infinity will be called $\text{Pred}^*(\Lambda)$.

For every $P \in \text{Pred}(\Lambda)$ we construct a $P^* \in \text{Pred}^*(\Lambda)$ as follows:

$$P^*(\omega, t, \omega', t') = P(\omega, t, \omega', t') \text{ if } t' < \infty, \text{ and}$$

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

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

112200
112250
112300
112350
112400
112450
112500
112550
112600
112650
112700
112750
112800
112850
112900
112950
113000
113050
113100
113150
113200
113250
113300
113350
113400
113450
113500
113550
113600
113650
113700
113750
113800
113850
113900
113950
114000
114050
114100
114150
114200
114250
114300
114350
114400
114450
114500
114550
114600
114650
114700
114750
114800
114850
114900
114950
115000
115050
115100
115150
115200
115250

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 $P, Q \in \text{Pred}(\Lambda)$.

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

$$\begin{aligned} \lambda^* ((\omega = \omega') \wedge (t' = \infty)) &= \lambda^* (t' = \infty) = \lambda (t' = \infty) = \text{NONTERM}, \\ \lambda^* ((\omega = \omega') \wedge (t = t')) &= \lambda ((\omega = \omega') \wedge (t = t')) \vee (t = t' = \infty) = \\ &= \lambda ((t = t') \wedge ((t' < \infty) \rightarrow (\omega = \omega'))) = \perp. \end{aligned}$$

If P and Q are in $\text{Pred}(\Lambda)$ 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 $P \in \text{Pred}(\Lambda)$

115300
115350
115400
115450
115500
115550
115600
115650
115700
115750
115800
115850
115900
115950
116000
116050
116100
116150
116200
116250
116300
116350
116400
116450
116500
116550
116600
116650
116700
116750
116800
116850
116900
116950
117000
117050
117100
117150
117200
117250
117300
117350
117400
117450
117500
117550
117600
117650
117700
117750
117800
117850
117900
117950
118000
118050
118100
118150
118200
118250
118300
118350

$P * J = P^*$,
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
 $J * J = J^*$.

3. Semantic information.

We assume that we have a set called $\text{Prog}(\Omega)$. 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 $\text{Prog}(\Omega)$ into $\text{Pred}^*(\Lambda)$. The interpretation is that if π
is a program, then $\text{Totinf}(\pi)$ 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 $\text{Totinf}(\pi)$.

Quite often it happens that for a given program π it
is hard to find $\text{Totinf}(\pi)$ 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 $R \in \text{Pred}^*(\Lambda)$ such that

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

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

118400
118450
118500
118550
118600
118650
118700
118750
118800
118850
118900
118950
119000
119050
119100
119150
119200
119250
119300
119350
119400
119450
119500
119550
119600
119650
119700
119750
119800
119850
119900
119950
120000
120050
120100
120150
120200
120250
120300
120350
120400
120450
120500
120550
120600
120650
120700
120750
120800
120850
120900
120950
121000
121050
121100
121150
121200
121250
121300
121350
121400
121450

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 $\text{Prog}(\Omega) \times \text{Pred}^*(\Lambda)$, 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
if $\text{Totinf}(\pi) \subset P$ then $w(\pi, P)$
for all $\pi \in \text{Prog}(\Omega), P \in \text{Pred}^*(\Lambda)$.

In the present note we shall not follow the line of [13].
The advantage of avoiding $\text{Totinf}(\pi)$ might not compensate the
disadvantage that the basic properties of $w(\pi, P)$ are harder
to formulate than those of $\text{Totinf}(\pi)$. 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 $\text{Totinf}(\pi)$ 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 $\text{Totinf}(\pi)(\omega, t, \omega', t')$ is true.
Nevertheless we do not formulate this as a general condition on
 $\text{Totinf}(\pi)$, 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 $\text{Prog}(\Omega)$ we write

$$\pi \leq \sigma \quad \text{iff} \quad \text{Totinf}(\pi) \subset \text{Totinf}(\sigma)$$

If both $\pi \leq \sigma$ and $\sigma \leq \pi$ we say that π and σ are semantically

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

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

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

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

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

$$\text{Totinf}(\text{adlibitum}) = \text{TRUE}.$$

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

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

The trouble with a formal treatment of assignments is that

124600
124650
124700
124750
124800
124850
124900
124950
125000
125050
125100
125150
125200
125250
125300
125350
125400
125450
125500
125550
125600
125650
125700
125750
125800
125850
125900
125950
126000
126050
126100
126150
126200
126250
126300
126350
126400
126450
126500
126550
126600
126650
126700
126750
126800
126850
126900
126950
127000
127050
127100
127150
127200
127250
127300
127350
127400
127450
127500
127550
127600
127650

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 how g is obtained from E; one usually has the "naive" interpretation that the value g(ω) is obtained if we just replace x by ω in E. But whatever the relation between E and g might be, the semantics is

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

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

$$P = \text{Totinf}^{\omega}(\pi), Q = \text{Totinf}^{\omega}(\sigma), R = \text{Totinf}^{\omega}(\pi \text{ or } \sigma), \text{ by}$$

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

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

127700 Next we consider the well-known construct " $\pi ; \sigma$ ", called
 127750 the concatenation of the programs π and σ . The semantics is
 127800 given by means of the convolution
 127850
 127900

$$Totin^2(\pi ; \sigma) = Totin^2(\pi) * Totin^2(\sigma).$$

128000 We note that $Totin^2(\pi ; \sigma)$ is constant at infinity, since the second
 128050 factor is constant at infinity. We also note that because of the
 128100 associativity of the convolution the programs $\pi ; (\sigma ; \tau)$ and
 128150 $(\pi ; \sigma) ; \tau$ are semantically equivalent.

128200 With the construct "if E then π else σ " we have a
 128250 situation similar to the one with the assignment statement
 128300 we considered in section 4. Somehow we have defined a class
 128350 of expressions in the programming language, and to each E of
 128400 that class we have associated a predicate B on Ω . The
 128450 expression E may contain a symbol x referring to the input
 128500 value ω , but no symbols referring to ω' , t or t' . (The usual
 128550 "naive" point of view is that the program text contains B itself,
 128600 and that it reads "if $B(x)$ then π else σ "). The intuitive
 128650 (operational) meaning of "if E then π else σ " is that if $B(\omega)$
 128700 is true then π is to be executed, if $B(\omega)$ is false then σ is
 128750 to be executed.

128800 The formal semantics is described as follows. Let P, Q, R be
 128850 the $Totin^2$'s of π, σ and "if E then π else σ ". Then we have

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

130000 and we note that the right-hand side is equivalent to
 130050
 130100 $(B(\omega) \rightarrow P(\omega, t, \omega', t')) \wedge (\neg B(\omega) \rightarrow Q(\omega, t, \omega', t'))$.

130150 We have to check that R is constant at infinity. This
 130200 follows directly from the fact that both P and Q are constant at
 130250 infinity.

130300 In the lower primitive program constructs of this section
 130350
 130400
 130450
 130500
 130550
 130600
 130650
 130700
 130750

130800
130850
130900
130950
131000
131050
131100
131150
131200
131250
131300
131350
131400
131450
131500
131550
131600
131650
131700
131750
131800
131850
131900
131950
132000
132050
132100
132150
132200
132250
132300
132350
132400
132450
132500
132550
132600
132650
132700
132750
132800
132850
132900
132950
133000
133050
133100
133150
133200
133250
133300
133350
133400
133450
133500
133550
133600
133650
133700
133750
133800
133850

we have not administered any run time for the "overhead" of the constructor: the only time consumption is in the execution of the sub-programs π, σ, \dots . 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 $\text{Prog}(\Omega)$ 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 $\pi \in \text{Prog}(\Omega)$ we define $\varphi(\pi)$ by

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

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

procedure φ ; $\varphi(\varphi)$.

In order to describe the semantics of $\text{RECURS}(\varphi)$ we first define the program-to-program function ψ by

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

Next we introduce the functions ψ^k ($k=0, 1, \dots$) 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

$$R_k = \text{Totinf}(\psi^k(\text{adlibitum})) \tag{2}$$

we now define the semantics of $\text{RECURS}(\varphi)$ by

$$\text{Totinf}(\text{RECURS}(\varphi)) = R \tag{3}$$

where

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

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

$$R_0 = \text{Totinf}(\text{adlibitum}) = \text{TRUE.}$$

133900 We have to check that $R \in \text{Pred}^*(\wedge)$, i.e., that R is constant
 133950 at infinity. This is trivial from (4), since each R_k is constant at
 134000 infinity.
 134050
 134100
 134150

134200 In section 9 we shall comment on the definition (3), and
 134250 discuss its relation to fixed point semantics. We postpone the
 134300 discussion since we want to show first that (3) is good enough for
 134350 a practical case: the "while" statement.
 134400
 134450
 134500
 134550
 134600
 134650
 134700
 134750

134800 7. The while statement
 134850

134900 We consider the program that is usually written as

134950 while E do τ (1)
 135000
 135050

135100 where τ is a program, and E is an expression that plays the
 135150 same role as in section 4 in the construct "if E then π else σ ".
 135200
 135250

135300 We form a program-to-program function φ by

135350 $\varphi(\pi) = \text{if } E \text{ then } (\tau; \pi) \text{ else skip}$ (2)
 135400
 135450

135500 and we claim that the semantics of $\text{RECURS}(\varphi)$ is able to explain
 135550 the semantics usually attached to (1). We shall do this both
 135600 for partial correctness and for total correctness. In both cases
 135650 B is the predicate on Ω that corresponds to E.
 135700
 135750
 135800
 135850
 135900
 135950

136000 Theorem 1 ("Partial correctness). Let C be a predicate on Ω

136050 and abbreviate

136100 $B = \lambda ((t=t') \wedge ((B(\omega) \wedge C(\omega)) \rightarrow C(\omega'))))$, (3)
 136150
 136200

136250 $F = \lambda ((C(\omega) \wedge (t' < \infty)) \rightarrow ((C(\omega') \wedge \neg B(\omega'))))$. (4)
 136300
 136350

136400 Assume that the semantics of τ satisfies

136450 $\text{Totinf}(\tau) \subset B^*$. (5)
 136500
 136550

136600 Then the one of $\text{RECURS}(\varphi)$ satisfies

136650 $\text{Totinf}(\text{RECURS}(\varphi)) \subset F$. (6)
 136700
 136750

136800 Proof. As in section 6 we abbreviate

136850 $R_k = \text{Totinf}(\psi^k(\text{adlibitum}))$,
 136900
 136950

and we note that

$$\psi^{k+1}(\text{adlibitum}) = \text{if } E \text{ then } (\tau; \text{delay}; \psi^k(\text{adlibitum})) \text{ else skip.}$$

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

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

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

$$S = \text{Totin}^2(\text{skip}) = \lambda^*((\omega = \omega') \wedge (t = t')),$$

By (3) we have

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

and therefore (8) can be simplified to

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

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

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

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

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

Once this has been shown we rapidly get to (6): by section 6

$\text{Totin}^2(\text{RECURS}(\varphi)) \subset R_k$ 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

137000
 137050
 137100
 137150
 137200
 137250
 137300
 137350
 137400
 137450
 137500
 137550
 137600
 137650
 137700
 137750
 137800
 137850
 137900
 137950
 138000
 138050
 138100
 138150
 138200
 138250
 138300
 138350
 138400
 138450
 138500
 138550
 138600
 138650
 138700
 138750
 138800
 138850
 138900
 138950
 139000
 139050
 139100
 139150
 139200
 139250
 139300
 139350
 139400
 139450
 139500
 139550
 139600
 139650
 139700
 139750
 139800
 139850
 139900
 139950
 140000
 140050

140100
140150
140200
140250
140300
140350
140400
140450
140500
140550
140600
140650
140700
140750
140800
140850
140900
140950
141000
141050
141100
141150
141200
141250
141300
141350
141400
141450
141500
141550
141600
141650
141700
141750
141800
141850
141900
141950
142000
142050
142100
142150
142200
142250
142300
142350
142400
142450
142500
142550
142600
142650
142700
142750
142800
142850
142900
142950
143000
143050
143100
143150

$R_0 \subset P_0$ (15)

is trivial: $P = \text{TRUE}$ since $t'+1 \leq t$ is false on \wedge . 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

$R_{k+1} \subset P_{k+1}$ (16)

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

$R_k \subset P_k$ and $R_{k+1}(\omega, t, \omega', t')$ (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

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

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

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

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

we derive

$(D(\omega) \wedge (t'+1 \leq k+t+1)) \rightarrow (D(\omega') \wedge \neg B(\omega'))$ (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 $D(\omega) \wedge (t'+1 \leq k+1+t) \wedge (t' < \infty)$

and want to show $D(\omega') \wedge \neg B(\omega')$. From $S(\omega, t, \omega', t')$ and

$t' < \infty$ we get $\omega = \omega'$, so by $D(\omega)$ and $\neg B(\omega)$ we have

$D(\omega') \wedge \neg B(\omega')$. This finishes the proof of Theorem 1.

Theorem 2 ("Total correctness"). Let C be a predicate on \mathcal{R} ,

and let Q be a mapping of \mathcal{R} into the set of integers ≥ 0 .

We abbreviate

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

143200
143250
143300
143350
143400
143450
143500
143550
143600
143650
143700
143750
143800
143850
143900
143950
144000
144050
144100
144150
144200
144250
144300
144350
144400
144450
144500
144550
144600
144650
144700
144750
144800
144850
144900
144950
145000
145050
145100
145150
145200
145250
145300
145350
145400
145450
145500
145550
145600
145650
145700
145750
145800
145850
145900
145950
146000
146050
146100
146150
146200
146250

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

Assume that the semantics of τ satisfies

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

Then the one of $\text{RECURS}(\varphi)$ satisfies

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

Proof. With the new D and F we follow the proof of

Theorem 1. We also take new P_k 's:

$$P_k = \lambda((D(\omega) \wedge (Q(\omega) < k) \wedge (t < \infty)) \rightarrow ((t'+1 \leq t+k) \wedge D(\omega') \wedge \neg B(\omega'))). \quad (26)$$

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 $t' = \infty$

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

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 $D(\omega) \wedge (t < \infty)$. Taking $k > Q(\omega) + 1$ we infer from $P_k(\omega, t, \omega', t')$ that $t'+1 \leq t+k$ and $D(\omega') \wedge \neg B(\omega')$ and therefore $(t' \leq t + Q(\omega)) \wedge D(\omega') \wedge \neg B(\omega')$. 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) $P_0 = \text{TRUE}$.

Next we take any $(\omega, t, \omega', t') \in \Lambda$ and any $k \in \{0, 1, \dots\}$ 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 $D(\omega) \wedge (Q(\omega) < k+1) \wedge (t < \infty)$, and try to get $(t' \leq t+k) \wedge D(\omega') \wedge (\neg B(\omega'))$. The $D^*(\omega, t, \sigma, t)$ of (20) equals $D(\omega, t, \sigma, t)$ since $t < \infty$, and therefore implies in the present context $D(\sigma) \wedge ((Q(\sigma) < Q(\omega)))$.

146300
146350
146400
146450
146500
146550
146600
146650
146700
146750
146800
146850
146900
146950
147000
147050
147100
147150
147200
147250
147300
147350
147400
147450
147500
147550
147600
147650
147700
147750
147800
147850
147900
147950
148000
148050
148100
148150
148200
148250
148300
148350
148400
148450
148500
148550
148600
148650
148700
148750
148800
148850
148900
148950
149000
149050
149100
149150
149200
149250
149300
149350

Hence $D(\sigma) \wedge (Q(\sigma) < k)$. The $P_k(\sigma, t+1, \omega', t')$ of (20) now gives $(t' \leq t+k) \wedge D(\omega') \wedge \neg B(\omega')$ 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 $D(\omega) \wedge (Q(\omega) < k+1) \wedge (t < \infty)$. If $t' < \infty$ then $S(\omega, t, \omega', t')$ says that $\omega = \omega'$ and $t = t'$, and we get $(t'+1 - t+k+1) \leq D(\omega') \wedge \neg B(\omega')$. 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 "while E do τ " is given by $B(\omega) = (\omega > 0)$. And τ is a program for which we assume $\text{TotInf}^2(\tau) \subset D^*$, where

$$D = \lambda((t = t') \wedge (\omega' = \omega + 1)). \tag{27}$$

(in ALGOL one can think of the while-statement "while $x > 0$ do $x := x + 1$ "). We shall derive the semantical statement $\text{TotInf}(\text{RECURS}(\varphi)) \subset F$, where

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

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

Again we follow the proof of Theorem 1. We take

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

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

149400
149450
149500
149550
149600
149650
149700
149750
149800
149850
149900
149950
150000
150050
150100
150150
150200
150250
150300
150350
150400
150450
150500
150550
150600
150650
150700
150750
150800
150850
150900
150950
151000
151050
151100
151150
151200
151250
151300
151350
151400
151450
151500
151550
151600
151650
151700
151750
151800
151850
151900
151950
152000
152050
152100
152150
152200
152250
152300
152350
152400
152450

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 $P_0 = \text{TRUE}$ (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 $H^*(\omega, t, \sigma, t)$ 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

152500 as always trouble with the metalanguages of SEM and SYN, in
 152550 particular in those cases where we use one and the same word
 152600 (like "variable") in different meanings in the two metalanguages.
 152650
 152700
 152750

152800 In spite of the formidable amount of knowledge
 152850 about formal languages it must be said that SYN is a
 152900 poor man's world, an underdeveloped country. SYN cannot
 152950 really live without SEM, but SEM can certainly live very
 153000 comfortably without SYN, just by developing a bit of extra
 153050 metalanguage.
 153100
 153150
 153200
 153250
 153300
 153350

153400 Let us compare the situation of computer programs
 153450 with a subject that came up about two thousand years
 153500 earlier: geometrical constructions with ruler and compass.
 153550
 153600
 153650

153700 In this geometrical case SEM is the world of geometrical
 153750 objects and logical discussions about those objects. Since
 153800 the whole of mathematics is available to SEM, it includes sets
 153850 and mappings. As an example we mention that there is a mapping
 153900 that attaches to each pair (P, r) , where P is a point and r a
 153950 line segment, the set $\text{circle}(P, r)$, which is the set of all points
 154000 in our plane which have the distance r to P . Let us call the act
 154050 of getting the set $\text{circle}(P, r)$ from P and r a construction.
 154100
 154150
 154200
 154250
 154300
 154350
 154400
 154450

154500 In the metalanguage we now describe sequences of such
 154550 constructs, which lead from a set of objects we are assumed
 154600 to "have" at the start, to the objects which somehow interest
 154650 us. We say that this sequence constructs these interesting
 154700 objects. Parallel to this sequence we have a sequence of actions
 154750 in our physical world on physical paper with physical ruler,
 154800 compass and pencil, and for this sequence of physical actions
 154850 the sequence in SEM is a mathematical model. But we have to
 154900 emphasize that the world of SEM is bigger than this. We might
 154950 study constructions for which no physical realization is
 155000 available.
 155050
 155100
 155150
 155200
 155250
 155300
 155350
 155400
 155450
 155500
 155550

155800
155850
155700
155750
155800
155850
155900
155950
156000
156050
156100
156150
156200
156250
156300
156350
156400
156450
156500
156550
156600
156650
156700
156750
156800
156850
156900
156950
157000
157050
157100
157150
157200
157250
157300
157350
157400
157450
157500
157550
157600
157650
157700
157750
157800
157850
157900
157950
158000
158050
158100
158150
158200
158250
158300
158350
158400
158450
158500
158550
158600
158650

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 that 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

158700
158750
158800
158850
158900
158950
159000
159050
159100
159150
159200
159250
159300
159350
159400
159450
159500
159550
159600
159650
159700
159750
159800
159850
159900
159950
160000
160050
160100
160150
160200
160250
160300
160350
160400
160450
160500
160550
160600
160650
160700
160750
160800
160850
160900
160950
161000
161050
161100
161150
161200
161250
161300
161350
161400
161450
161500
161550
161600
161650
161700
161750

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 representably 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

161800
161850
161900
161950
162000
162050
162100
162150
162200
162250
162300
162350
162400
162450
162500
162550
162600
162650
162700
162750
162800
162850
162900
162950
163000
163050
163100
163150
163200
163250
163300
163350
163400
163450
163500
163550
163600
163650
163700
163750
163800
163850
163900
163950
164000
164050
164100
164150
164200
164250
164300
164350
164400
164450
164500
164550
164600
164650
164700
164750
164800
164850

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 "Totin?", 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 $\text{Prog}(\Omega)$ is representable in SYN, and we do not require that two elements of $\text{Prog}(\Omega)$ are equal if they have the same semantics.

Note that sometimes the notion of equality in SYN might be stronger than the one in $\text{Prog}(\Omega)$. 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 $\text{Prog}(\Omega)$ 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 $\text{Prog}(\Omega)$, but yet semantically equivalent.

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

identifying it with the set of all relations, like $\text{Pred}(\Delta)$.

9. Comments on the semantics of recursion.

In section 6 ~~we~~ 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

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

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 ,

168000
168050
168100
168150
168200
168250
168300
168350
168400
168450
168500
168550
168600
168650
168700
168750
168800
168850
168900
168950
169000
169050
169100
169150
169200
169250
169300
169350
169400
169450
169500
169550
169600
169650
169700
169750
169800
169850
169900
169950
170000
170050
170100
170150
170200
170250
170300
170350
170400
170450
170500
170550
170600
170650
170700
170750
170800
170850
170900
170950
171000
171050

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 $\text{Totinf}(\varphi^k(\mu))$ 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 $\text{Totinf}(\varphi^k(\mu))$ 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 $\mu, \varphi(\mu), \varphi^2(\mu), \dots$ 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: $\text{adlibitum} \geq \varphi(\text{adlibitum})$, and if φ is monotonic this leads to

$$\text{adlibitum} \geq \varphi(\text{adlibitum}) \geq \varphi^2(\text{adlibitum}) \geq \dots$$

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 $\text{Totinf}(\varphi^k(\text{adlibitum}))$ is just $\text{Totinf}(\text{adlibitum})$. This means that the sequence provides no semantic information at all: it just says that anything may happen, and not that $\text{RECURS}(\varphi)$ 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 ψ

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

171100
171150
171200
171250
171300
171350
171400
171450
171500
171550
171600
171650
171700
171750
171800
171850
171900
171950
172000
172050
172100
172150
172200
172250
172300
172350
172400
172450
172500
172550
172600
172650
172700
172750
172800
172850
172900
172950
173000
173050
173100
173150
173200
173250
173300
173350
173400
173450
173500
173550
173600
173650
173700
173750
173800
173850
173900
173950
174000
174050
174100
174150

Taking limits for $k \rightarrow \infty$ 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

$$\pi_1 \geq \pi_2 \geq \pi_3 \geq \dots \tag{1}$$

We say that

$$\pi_k \rightarrow \pi \quad (k \rightarrow \infty) \tag{2}$$

if for all (ω, t, ω', t')

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

where $P_k = \text{Totinf}(\pi_k)$, $P = \text{Totinf}(\pi)$. We note that $\text{Totinf}(\pi)$ is uniquely determined by the sequence π_1, π_2, \dots , and moreover that

$$\pi_k \geq \pi \tag{3}$$

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 $\text{Totinf}(\pi_k)$ as information about π , since (3) expresses $\text{Totinf}(\pi_k) \geq \text{Totinf}(\pi)$. This is much simpler than with lim sup, where we can obtain information about the lim sup only if we have information about $\text{Totinf}(\pi_k)$ for infinitely many k .

Let us now discuss the idea of $\text{RECURS}(\varphi)$ being a fixed point. Intuitive ideas about execution suggest that the "fixed point statement"

$$\text{Totinf}(\varphi \text{ RECURS}(\varphi)) = \text{Totinf}(\text{RECURS}(\varphi)), \tag{4}$$

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:

$$\psi(\text{RECURS}(\varphi)) \leq \text{RECURS}(\varphi). \quad (5) \quad (27)$$

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

$$\text{RECURS}(\varphi) \leq \psi^k \text{ (ad libitum)}$$

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

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

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

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 φ 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 π_k with $\text{Totinf}(\pi_k) = \lambda((\omega' \geq k) \wedge (t=t') \vee (t'=\infty))$.
 No less σ be described simply by $\text{Totinf}(\sigma) = \lambda((\omega'=0) \wedge (t=t'))$.

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

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

for all τ .

We have $\pi_1 \geq \pi_2 \geq \dots$. If we call the limit π it follows from the monotonicity of φ that $\varphi(\pi_1) \geq \varphi(\pi_2) \geq \dots$

Let us put $\rho = \lim \varphi(\pi_k)$. We hope that ρ and $\theta(\pi)$ are semantically equivalent, but unfortunately this is not the case:

$$\begin{aligned} \text{Totinf}(\rho) &= \lambda^*((\omega'=0) \wedge (t=t')) \vee (t'=\infty) \\ \text{Totinf}(\theta(\pi)) &= \lambda^*(t'=\infty). \end{aligned}$$

177300
177350
177400
177450
177500
177550
177600
177650
177700
177750
177800
177850
177900
177950
178000
178050
178100
178150
178200
178250
178300
178350
178400
178450
178500
178550
178600
178650
178700
178750
178800
178850
178900
178950
179000
179050
179100
179150
179200
179250
179300
179350
179400
179450
179500
179550
179600
179650
179700
179750
179800
179850
179900
179950
180000
180050
180100
180150
180200
180250
180300
180350

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 than 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 $\text{Totinf}(\psi(\pi)) = F(\text{Totinf}(\pi))$ for all π . Then if $P = \text{Totinf}(\text{RECURS}(\psi))$ 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: F is the maximal fixed point of F . In order to show $Q \subset P$ we remark that

$$\text{TRUE} \supset F(\text{TRUE}) \supset F^2(\text{TRUE}), \dots$$

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

$$Q, F(Q), F^2(Q), \dots$$

(of which all entries equal Q) we infer from $Q \subset \text{TRUE}$, by monotonicity of F , that $Q \subset F^k(\text{TRUE})$, 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 $\lim(\text{Totinf}(\psi^k(\text{adlibitum})))$ and any definition based on the idea of a maximal fixed point.

180400
180450
180500
180550
180600
180650
180700
180750
180800
180850
180900
180950
181000
181050
181100
181150
181200
181250
181300
181350
181400
181450
181500
181550
181600
181650
181700
181750
181800
181850
181900
181950
182000
182050
182100
182150
182200
182250
182300
182350
182400
182450
182500
182550
182600
182650
182700
182750
182800
182850
182900
182950
183000
183050
183100
183150
183200
183250
183300
183350
183400

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 $Totinf(\omega, t, \omega', t')$ 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

$$\underline{f_1} \ E_1 \rightarrow \pi_1 \ \square \ \dots \ \square \ E_k \rightarrow \pi_k \ \underline{f_2} \tag{1}$$

where each E_i corresponds to a predicate B (like in our discussion of the "if then else" 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

$$(B_1(\omega) \wedge P_1(\omega, t, \omega', t')) \vee \dots \vee (B_k(\omega) \wedge P_k(\omega, t, \omega', t')), \tag{2}$$

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

183450
183500
183550
183600
183650
183700
183750
183800
183850
183900
183950
184000
184050
184100
184150
184200
184250
184300
184350
184400
184450
184500
184550
184600
184650
184700
184750
184800
184850
184900
184950
185000
185050
185100
185150
185200
185250
185300
185350
185400
185450
185500
185550
185600
185650
185700
185750
185800
185850
185900
185950
186000
186050
186100
186150
186200
186250
186300
186350
186400
186450
186500

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

$$\text{Totinf}(\pi)(\omega, t, \omega', t') = ((\omega' = b) \vee (\omega' = c)) \wedge (t = t')$$
$$\text{Totinf}(\sigma)(\omega, t, \omega', t') = (\omega \neq b) \wedge (\omega = \omega') \wedge (t = t').$$

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

$$\text{Totinf}(\pi; \sigma)(\omega, t, \omega', t') = (\omega' = c) \wedge (t' = t) \quad (3)$$

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 " $ab := \text{false}$ " (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 $ab = \text{true}$ 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 $(1/p) \geq m$ then $y := 1/p$ else $ab := \text{true}$ ".

186650
186600
186650
186700
186750
186800
186850
186900
186950
187000
187050
187100
187150
187200
187250
187300
187350
187400
187450
187500
187550
187600
187650
187700
187750
187800
187850
187900

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 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 of the mathematical language AUTOMATH. Report, Department of mathematics, Technological University Eindhoven, The Netherlands. November 1973.

The two kinds of semantics get closer together if we take a more liberal interpretation of non-termination. In this more liberal version a semantical statement that says (with ω, t given), that $t' = \infty$ is a possible effect, has to be interpreted as that there is no upper bound to the values of the t' of the possible executions (with initial values ω, t). After all, if we are interested in having our programs terminated, a statement that a program might run for a million years does not give us much more comfort than a statement that it might go on for ever. Therefore it is quite reasonable to identify "unpredictably long" with "infinitely long".

This "liberal version" is related to the following definition. If $P \in \text{Pred}(\Lambda)$ then instead of the P^* of section 2 we define P^+ by: $P^+(\omega, t, \omega', t') = P(\omega, t, \omega', t')$ if $t' < \infty$, and

$$P^+(\omega, t, \omega', \infty) = \bigvee_{u \in T \setminus \{\infty\}} \exists v \in T \mid v > u \quad \exists \rho \in \Omega \quad P(\omega, t, \rho, v).$$

If the set of all P^+ with $P \in \text{Pred}(\Lambda)$ is denoted by $\text{Pred}^+(\Lambda)$, we have

$$\text{Pred}^+(\Lambda) \subset \text{Pred}^*(\Lambda) \subset \text{Pred}(\Lambda).$$

In deterministic cases there is no difference between P^* and P^+ . To be more precise, if ω, t are such that there is at most one pair ω', t' with $t' < \infty$ and $P(\omega, t, \omega', t')$, then $P^+(\omega, t, \omega', t') = P^*(\omega, t, \omega', t')$ for all ω', t' .

In general, if P describes the semantics of a program in the original version (where $P(\omega, t, \omega', \infty)$ means that the program can actually run for ever), then P^+ is the liberal semantics (where $P^+(\omega, t, \omega', \infty)$ means that there is no

103050
103100
103150
103200
103250
103300
103350
103400
103450
103500
103550
103600
103650
103700
103750
103800
103850
103900
103950
104000
104050
104100
104150
104200
104250
104300
104350
104400
104450
104500
104550
104600
104650
104700
104750
104800
104850
104900
104950
105000
105050
105100
105150
105200
105300
105350
105400
105450
105500
105550
105600
105700
105750
105800
105850
105900
105950
106050
106100
106150
106200
106250

upper bound to the runtime).

If we take the liberal semantics, then our prospects for proving the fixed point property for recursion become much more promising.

The difference between P^* and P^+ , and its being related to having non-determinism and infinite state space, can be connected to König's well known infinity lemma. In order to make the notation sufficiently clear for further discussion, we explain it in a few words.

Let (S,r,f) be a rooted tree. That is, S is a set (the set of "points"), r is a special element of S (called the "root") and f is a mapping of $S \setminus \{r\}$ into S ($f(x)$ is called the "father" of x). It is assumed that for every x ($x \neq r$) there is an integer n such that the n -th iterate f^n maps x into r . This n is uniquely determined, and is called the "level" of x . The level of r is zero. If $x \in S$, the set of all $y \in S \setminus \{r\}$ with $f(y)=x$ is called the "offspring" of x , and denoted $O(x)$.

If $x \in S$ we denote by $IP(x)$ the proposition that there is an infinite path starting from x , that is a sequence x_0, x_1, x_2, \dots with $x_0 = x$ and $f(x_{n+1}) = x_n$ for $n = 0, 1, 2, \dots$. And by $UL(x)$ (UL abbreviates "unbounded level") we denote the proposition that for every natural number m there is a path x_0, \dots, x_m , again with $x_0 = x$, $f(x_{n+1}) = x_n$ for $n = 0, \dots, m-1$.

We note that for all x

$$IP(x) \rightarrow UL(x).$$

Finally we formulate the "finite offspring condition". It says that for every $x \in S$ the offspring $O(x)$ is a finite

106300
106350
106400
106450
106500
106550
106600
106650
106700
106750
106800
106850
106900
106950
107000
107050
107100
107150
107200
107250
107300
107350
107400
107450
107500
~~107550~~
107650
107700
107750
107800
107850
107900
107950
108000
108050
108100
108150
108200
108300
108350
108400
108450
~~108550~~
108600
108650
108700
108750
108800
108850
108900
108950
109000
109050
109100
109150
109200
109250
109300
109350
109400
109450
109500

set.

König'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.

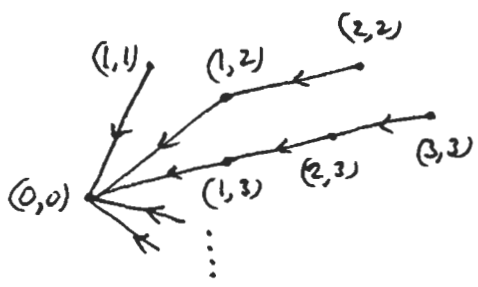


Figure 1

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

109550 Totinf(step) = (t'=t) ^ (ω' ∈ D(ω)).

109600 Note that "step" is a non-deterministic program, at least
 109650 if there exist points x where D(x) has more than one element.
 109700 If D(ω) is empty (such an x is called an end-point), then
 109750 there is no possible output ω' to the input ω. If this is
 109800 considered unacceptable, one might take any arbitrary value
 109850 of ω' as output, like ω'=ω. That means that in the definition
 109900 of Totinf(step) we replace ω' ∈ D(ω) by
 109950 ω' ∈ D(ω) ∨ ((D(ω) = ∅) ^ (ω'=ω)).

110000 ω' ∈ D(ω) ∨ ((D(ω) = ∅) ^ (ω'=ω)).

110050 But actually the case of empty D(ω) is unimportant since the
 110100 program "step" will not be used there.

110150 Let us now discuss the program (see the beginning of
 110200 section 7)

110250 while E do step,

110300 where E corresponds to the predicate B given by

110350 B(ω) = (D(ω) ≠ ∅).

110400 In the notation of formula (2) of section 7 this program
 110450 denotes RECURS(φ), where φ is given by

110500 φ(π) = if E then (step ; π) else skip.

110550 The "intuitive", or, if one prefers, "operational" semantics
 110600 of this program is the following one. Let ω (the input) be any point
 110650 of the tree. If ω' is an end-point such that f^k(ω') = ω for
 110700 some k ≥ 0 then ω' is a possible output (and t'=t+k). If
 110750 IP(ω) holds then there is a non-terminating execution. If
 110800 IP(ω) is false but UL(ω) still holds then there exist unpredictably
 110850 long executions. If UL(ω) does not hold then all executions
 110900 starting with ω terminate, and there is an upper bound to their
 110950 runtime.

111000 Let us now investigate Totinf(RECURS(ω)) as defined
 111050 by (4) in section 6. As far as terminating executions are
 111100 concerned, it produces the same results as the intuitive

112750 semantics. For values of ω where $IP(\omega)$ holds, it proclaims
 112800 the possibility that $t' = \infty$, as it should. But in points where
 112850 $UL(\omega)$ holds but $IP(\omega)$ does not, the semantics of section 6 still
 112900 says that $t' = \infty$ is possible, and the intuitive semantics says it
 112950 is not.
 113000
 113050

113100 The difference between these two kinds of semantics vanishes
 113150 (at least for this program) if we identify "unpredictably long
 113200 runtime" and "non-termination".
 113250

113300 In the tree program described here, it is also easy to
 113350 illustrate that $\psi(RECURS(\varphi))$ and $RECURS(\psi)$ can have different
 113400 semantics in the system of section 6. We take
 113450 ~~semantics. We take the special tree~~ (S_0, r_0, f_0) . We have $UL(r_0)$,
 113500 but $UL(x)$ is false for all x with $f(x) = r_0$. From this it can be
 113550 derived that at $(r_0, t, \omega', \infty)$ (where ω' irrelevant),
 113600 $RECURS(\varphi)$ is true but $\psi(RECURS(\varphi))$ is not.
 113650

113700 *Things look much better in the more liberal $Pred^+$ seman-*
 113750 *tics. We briefly discuss the changes that have to be made.*
 113800

113850 \bar{F}
 113900 First, in section 2 we have to introduce $Pred^+(\wedge)$ instead
 113950 of $Pred^*(\wedge)$. We have to give up the idea of a J such that always
 114000 $P * J = P^+$, like in formula (2) of section 2, and it is not
 114050 generally true that $(P * Q)^+ = P * Q^+$ (A counterexample:
 114100 $\Omega = \mathbb{N}$, $P(\omega, t, \omega', t') = (t' = t)$, $Q(\omega, t, \omega', t') =$
 114150 $= ((\omega' = \omega) \wedge t' = t + \omega)$).
 114200

114250 ~~In section 5 the semantics of the concatenation has to be des-~~
 114300 ~~cribed by~~
 114350

114400 ~~$Totinf(\pi ; \sigma) = (Totinf(\pi) * Totinf(\sigma))$~~

114450 In section 3 we have to introduce $Totinf^+$ as a mapping
 114500 of $Prog(\Omega)$ into $Pred^+(\wedge)$. The changes in section 4 are
 114550 trivial.
 114600

114650 In section 5 the semantics of the concatenation has to be
 114700 described by
 114750

114800 $Totinf(\pi ; \sigma) = (Totinf(\pi) * Totinf(\sigma))^+$

115850
115900
115950
116000
116050
116100
116150
116200
116250
116300
116400
116450
116500
116600
116650
116700
116750
116800
116850
116900
116950
117000
117050
117100
117150
117200
117250
117300
117350
117400
117450
117500
117550
117600
117650
117700
117750
117800
117850
117900
117950

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 $P_0 \supseteq P_1 \supseteq P_2 \supseteq \dots$ with

$$P_n = P_n^+ \text{ for all } n, \text{ and if } P = \bigvee_n P_n, \text{ then } P = P^+.$$

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 $\psi(\text{RECURS}(\varphi))$ and $\text{RECURS}(\varphi)$ 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 $\text{RECURS}(\varphi)$ can be considered as the maximal fixed point.