A Simple Axiomatic Basis for Programming Language Constructs.

by Edsger W.Dijkstra

Abstract. The semantics of a program can be defined in terms of a predicate transformer associating with any post-condition (characterizing a set of final states) the corresponding weakest pre-condition (characterizing a set of initial states). The semantics of a programming language can be defined by regarding a program text as a prescription for constructing its corresponding predicate transformer.

Its conceptual simplicity, the modest amount of mathematics needed and its constructive nature seem to be its outstanding virtues. In comparison with alternative approaches it should be remarked, firstly, that all non-terminating computations are regarded as equivalent and, secondly, that a program construct like the goto-statement falls outside its scope; the latter characteristic, however, does not strike the author as a shortcoming, on the contrary, it confirms him in one of his prejudices!

A Simple Axiomatic Basis for Programming Language Constructs.

by Edsger W.Dijkstra

Program testing can be used very effectively to show the presence of bugs, but is hopelessly inadequate for showing their absence and a convincing correctness proof seems the only way to reach the required confidence level.

In order that such a convincing correctness proof may exist, two conditions must be satisfied by such a correctness proof:
1) it must be a proof and that implies that we need a set of axioms to start with
2) it must be convincing and that implies that we must be able to write, to check, to understand and to appreciate the proof.
This essay deals with the first of these two topics.

We are considering finite computations only; therefore we can restrict ourselves to computational processes taking place in a finite state machine -although the possible number of states may be very, very large- and take the point of view that the net effect of the computation can be described by the transition from initial to final state. (Since the classical work of A.M.Turing, and again since the recent work of D.Scott, one often encounters the tacid assumption that size and speed of today's computers are so huge, that the inclusion of infinite computations leads to the most appropriate model. I would be willing to buy that if -as in the case of "the line at infinity", sweetly leading to projective geometry- the suggested generalization would clean up the majority of the arguments. Thanks to Cantor, Dedekind et al., however, we know that the inclusion of the infinite computation is <u>not</u> a logically painless affair, on the contrary! In the light of that experience it seems more effective to restrict oneself to finite computations taking place in a finite, but sufficiently large universe, thereby avoiding a number of otherwise self-inflicted pains. Those mathematicians that are so biased as to refuse to consider an area of thought worthy of their attention, unless it pays full attention to their pet generalizations, should perhaps not try to follow the rest of my argument.) The computation is assumed to take place under control of an algorithm, and we want to make assertions about all possible computations that may be evoked under control of such a program.

And we want to base these assertions on the program text! (In particular for sub-programs our aims are usually more modest, being content with assertions about the class of computations that can take place under the additional constraint that the initial state satisfies some further condition, as we are able to show that such a condition will always be satisfied whenever the sub-program is invoked.)

This implies that we must have a formal definition of the semantics of the programming language in which the program has been expressed.

The earliest efforts directed towards such definition of semantics that I am aware of have been what I call "mechanistic definitions": they gave a definition (or "a description") of the steps that should be carried out in executing a program, they gave you "the rules of the game" necessary to carry out any given computation (as determined by program and initial state!) by hand or by machine. The basic shortcoming of this approach was that the semantics of an algorithm were expressed in terms of "the rules of the game", i.e. in terms of another algorithm. The game can only be played for a chosen initial state, and as a result it is as powerless as program testing! A mechanistic definition as such is not a sound basis for making assertions about the whole class of possible computations associated with a program. It is this shortcoming that the axiomatic method seeks to remedy.

We consider predicates P, Q, R, ... on the set of states; for each possible state a given predicate will be either true or false and if we so desire, we can regard the predicate as characterizing the subset of states for which it is true. There are two special predicates, named T and F: T is true for all possible states (characterizes the universe), F is false for all possible states (characterizes the empty set). We call two predicates P and Q equal ("$P = Q$") when the sets of states for which they are true are the same. (Note that $P \neq T$ -or non($P = T$)- does not allow us to conclude $P = F$!) We restrict ourselves to state spaces that are defined as the cartesian product of (the individual state spaces of) a number of named variables of known types. Predicates P, Q, R, ... are then formal expressions in terms of

1) the aforementioned variables (i.e. the "co-ordinates" of our state space)

2) constants of the appropriate types

3) free variables of the appropriate types.

The rules for evaluation of these formal expressions fall outside the scope of this essay: we assume them to be given "elsewhere", not tempted to redo, say, the work of a Boole or a Peano. (The ability to formulate the specifications to be met by the program presupposes that such work has already been done "elsewhere".)

We consider the semantics of a program S fully determined when we can derive for any post-condition P to be satisfied by the final state, the weakest pre-condition that for this purpose should be satisfied by the initial state. We regard this weakest pre-condition as a function of the post-condition P and denote it by "fS(P)". Here we regard the fS as a "predicate transformer", as a rule for deriving the weakest pre-condition from the post-condition to which it corresponds.

The semantics of a program S are defined when its corresponding predicate transformer fS is given, the semantics of a programming language are defined when the rules are given which tell how to construct the predicate transformer fS corresponding to any program S written in that language.

As most programming languages are defined recursively, we can expect such construction rules for the predicate transformer of the total program to be expressed in terms of predicate transformers associated with components. But, as we shall see in a moment, we must observe some restrictions, for if we allow ourselves too much freedom in the construction of predicate transformers we may arrive at predicate transformers fS such that fS(P) can no longer be interpreted as the weakest pre-condition corresponding to the post-condition P for a possible deterministic machine.

Our construction rules for predicate transformers fS must be such that, whatever fS we construct, it must have the following four basic properties:

1) P = Q implies fS(P) = fS(Q)
2) fS(F) = F
3) fS(P <u>and</u> Q) = fS(P) <u>and</u> fS(Q)
4) fS(P <u>or</u> Q) = fS(P) <u>or</u> fS(Q)

Predicate transformers enjoying those four properties we call "healthy".

Property 1 assures that we are justified in regarding the predicates as characterizing our true subject matter, viz. sets of states: it would be awkward if $fS(x > 0)$ differed from $fS(0 < x)$!

Property 2 is the so-called "Law of the Excluded Miracle" and does not need any further justification.

The justification for properties 3 and 4 becomes fairly obvious when we consider, for instance, $P = (0 \leq x \leq 2)$ and $Q = (1 \leq x \leq 3)$ and require that each initial state satisfying $fS(P)$ is mapped into a single state satisfying P and similarly for Q. Conversely it can be shown that each healthy predicate transformer fS can be interpreted as describing the net effect of a deterministic machine, whose actions are fully determined by the initial state.

From our 1st and 4th properties we can derive a conclusion. Let $P \Rightarrow Q$; from this it follows that there exists a predicate R such that we can write $Q = P \underline{or} R$. Our 1st and 4th properties then tell us that

$$fS(Q) = fS(P \underline{or} R) = fS(P) \underline{or} fS(R)$$

from which we deduce that

5) $P \Rightarrow Q$ implies $fS(P) \Rightarrow fS(Q)$.

A further useful property of healthy predicate transformers can be derived already at this stage. Properties 1 and 4 allow us to conclude for any P

$$fS(P) \underline{or} fS(\underline{non} P) = fS(P \underline{or} \underline{non} P) = fS(T) \quad .$$

Taking at both sides the conjunction with $\underline{non}$ $fS(P)$ we reach

$$fS(\underline{non} P) \underline{and} \underline{non} fS(P) = fS(T) \underline{and} \underline{non} fS(P) \quad .$$

Properties 1, 2 and 3 allow us to conclude for the same fS and same P

$$fS(P) \underline{and} fS(\underline{non} P) = fS(P \underline{and} \underline{non} P) = fS(F) = F \quad .$$

Taking in the last two formulae at both sides the disjunction we find for healthy predicate transformers property

6) $fS(\underline{non} P) = fS(T) \underline{and} \underline{non} fS(P)$

or, replacing P by $\underline{non}$ P and taking the negation at both sides, its alternative formulation

6') $\underline{\text{non}}\ fS(P) = fS(\underline{\text{non}}\ P)\ \underline{\text{or}}\ \underline{\text{non}}\ fS(T)$.

The simplest predicate transformer enjoying the four basic properties is the identity transformation:

$$fS(P) = P \quad .$$

The corresponding statement is well known to programmers, they usually call it "the empty statement".

But it is very hard to build up very powerful programs from empty statements alone, we need something more powerful. We really want to transform a given predicate P into a possibly different predicate fS(P).

One of the most basic operations that can be performed upon formal expressions is substitution, i.e. replacing all occurences of a variable by (the same) "something else". If in the predicate P all occurrences of the variable "x" are replaced by (E), then we denote the result of this tranformation by

$$P_{E \to x} \quad .$$

Now we can consider statements S such that

$$fS(P) = P_{E \to x} \quad ,$$

where x is a "co-ordinate variable" of our state space and E an expression of the appropriate type. The above rule introduces a whole class of statements, each of them given by three things

a) the identity of the variable x to be replaced

b) the fact that the substitution is the corresponding rule for predicate transformation

c) the expression E which is to replace every occurrence of x in P.

The usual way to write such a statement is

```
x := E
```

and such a statement is known under the name of an "assignment statement". We can formulate the

<u>Axiom of Assignment</u>. When the statement S is of the form $x := E$, its semantics are given by the predicate transformer fS that is such that for all P $\quad fS(P) = P_{E \to x}$.

The subsitution process leads to healthy predicate transformers.

Although from a logical point of view unnecessary -we can take this predicate transformer to give <u>by definition</u> the semantics of what we call assignment statements- it is wise to confront this axiomatic definition with our intuitive understanding of the assignment statement -if we have one!- and it is comforting to discover that indeed it captures the assignment statement as we (may) know it, as the following examples show. They are written in the format: $\{fS(P)\}\ S\ \{P\}$.

$$\{a > 0\} \quad x{:=}\ 1 \quad \{a > 0\}$$
$$\{(1) < 2\} \quad x{:=}\ 1 \quad \{x < 2\}$$
$$\{a > 0 \ \underline{\text{and}}\ (x + 1) < 9\} \quad x{:=}\ x + 1 \quad \{a > 0\ \underline{\text{and}}\ x < 9\} \quad .$$

The above rules enable us to establish the semantics of the empty program and of the program consisting of a single assignment statement. In order to be able to compose more complicated predicate transformers, we observe that the functional composition of two healthy predicate transformers is again healthy. So this is a legitimate way of constructing a new one and we are led to the

<u>Axiom of Concatenation</u>. Given two statements S1 and S2 with healthy predicate transformers fS1 and fS2 respectively, the predicate transformer fS, given for all P by

$$fS(P) = fS1(fS2(P))$$

is healthy and taken as the semantic definition of the statement S that we denote by $\quad S1\ ;\ S2 \quad$.

Functional composition is associative and we are therefore justified in the use of the term "concatenation": it makes no difference if we parse "S1 ; S2 ; S3" either as "(S1 ; S2); S3" or as "S1 ;(S2 ; S3)".

Relating the axiomatic definition of the concatenation operator ";" to our intuitive understanding of a sequential computation, it just means that each execution of S1 (when completed) will immediately be followed by an execution of S2 and, conversely, that each execution of S2 has immediately been preceded by an execution of S1. The functional composition identifies the initial state of S2 with the final state of S1.

Looking for new programming language constructs implies looking for new ways of constructing predicate transformers, but all this, of course, subject to the restriction that the ensuing predicate transformer must be healthy. And a number of obvious suggestions must be rejected on that ground, such as:

$$fS(P) = \underline{\text{non}}\ fS1(P)$$

for that would violate the Law of the Excluded Miracle.

Also

$$fS(P) = fS1(P)\ \underline{\text{and}}\ fS2(P)$$

must be rejected as such a fS violates the basic property 4:

$$\begin{aligned} fS(P\ \underline{\text{or}}\ Q) &= fS1(P\ \underline{\text{or}}\ Q)\ \underline{\text{and}}\ fS2(P\ \underline{\text{or}}\ Q) \\ &= \{fS1(P)\ \underline{\text{or}}\ fS1(Q)\}\ \underline{\text{and}}\ \{fS2(P)\ \underline{\text{or}}\ fS2(Q)\} \end{aligned}$$

while

$$fS(P)\ \underline{\text{or}}\ fS(Q) = \{fS1(P)\ \underline{\text{and}}\ fS2(P)\}\ \underline{\text{or}}\ \{fS1(Q)\ \underline{\text{and}}\ fS2(Q)\}$$

and they are in general different, as the first of the two leads to the additional terms in the disjunction

$$\{fS1(P)\ \underline{\text{and}}\ fS2(Q)\}\ \underline{\text{or}}\ \{fS1(Q)\ \underline{\text{and}}\ fS2(P)\} \quad .$$

Similarly, if we choose

$$fS(P) = fS1(P)\ \underline{\text{or}}\ fS2(P)$$

property 3 is violated, because

$$\begin{aligned} fS(P\ \underline{\text{and}}\ Q) &= fS1(P\ \underline{\text{and}}\ Q)\ \underline{\text{or}}\ fS2(P\ \underline{\text{and}}\ Q) \\ &= \{fS1(P)\ \underline{\text{and}}\ fS1(Q)\}\ \underline{\text{or}}\ \{fS2(P)\ \underline{\text{and}}\ fS2(Q)\} \end{aligned}$$

while

$$fS(P)\ \underline{\text{and}}\ fS(Q) = \{fS1(P)\ \underline{\text{or}}\ fS2(P)\}\ \underline{\text{and}}\ \{fS1(Q)\ \underline{\text{or}}\ fS2(Q)\}$$

and here the second one leads to the additional terms in the disjunction

$$\{fS1(P)\ \underline{\text{and}}\ fS2(Q)\}\ \underline{\text{or}}\ \{fS1(Q)\ \underline{\text{and}}\ fS2(P)\} \quad .$$

This leads to the suggestion that we look for fS1 and fS2 (in general fS_i) such that for any P and Q

$$i \neq j \quad \text{implies} \quad fS_i(P)\ \underline{\text{and}}\ fS_j(Q) = F \quad .$$

Doing it for a pair leads to the

<u>Axiom of Binary Selection</u>. Given two statements S1 and S2 with healthy predicate transformers fS1 and fS2 respectively and a predicate B, the predicate transformer fS, given for all P by

$$fS(P) = \{B \text{ \underline{and} } fS1(P)\} \text{ \underline{or} } \{\text{\underline{non} } B \text{ \underline{and} } fS2(P)\}$$

is healthy and taken as the semantic definition of the statement S that we denote by

$\underline{\text{if}}$ B $\underline{\text{then}}$ S1 $\underline{\text{else}}$ S2 $\underline{\text{fi}}$.

(This is readily extended to a choice between three, four or any explicitly enumerated set of mutually exclusive alternatives, leading to the so-called case-construction.)

For an arbitrary given sequence fS_i we can not hope that $i \neq j$ implies $fS_i(P)$ $\underline{\text{and}}$ $fS_j(Q) = F$ for any P and Q, but we may hope to achieve this if we can generate the fS_i by a recurrence relation. Before we embark upon such a project, however, we should derive a useful property of the predicate transformers we have been willing to construct thus far.

If two predicate transformers fS and fS' satisfy the property that for all P: $fS(P) \Rightarrow fS'(P)$, then we call fS as strong as fS' and fS' as weak as fS.

(The predicate transformer given for all P by $fS(P) = F$ is as strong as any other, the predicate transformer given by $fS(P) = T$ would be as weak as any other if it were admitted, but it is not healthy: it violates the Law of the Excluded Miracle.)

We can now formulate and derive our

<u>Theorem of Monotonicity</u>. Whenever in a predicate transformer fS, formed by concatenation and/or selection, one of the constituent predicate transformers is replaced by one as weak (strong) as the original one, the resulting predicate transformer fS' is as weak (strong) as fS.

Obviously we only need to show this for the elementary transformer constructions.

Concatenation, case 1:

Let S be: S1 ; S2

let S' be: S1'; S2

let S1' be as weak as S1,

then for any P, $fS(P) = fS1(Q)$ and $fS'(P) = fS1'(Q)$, with $Q = fS2(P)$; as $fS1(Q) \Rightarrow fS1'(Q)$ for any Q, $fS(P) \Rightarrow fS'(P)$ for any P. QED.

Concatenation, case 2:

Let S be: S1 ; S2

let S' be: S1 ; S2'

let S2' be as weak as S2,

then for any P, $fS(P) = fS1(Q)$ and $fS'(P) = fS1(R)$ where $Q = fS2(P)$ and $R = fS2'(P)$. Because for any P, $Q \Rightarrow R$, it follows from the healthiness of fS1, that $fS(P) \Rightarrow fS'(P)$ for any P. QED.

Binary selection, case 1:

Let S be: <u>if</u> B <u>then</u> S1 <u>else</u> S2 <u>fi</u>

let S' be: <u>if</u> B <u>then</u> S1' <u>else</u> S2 <u>fi</u>

let S1' be as weak as S1,

then for any P

$$fS(P) = \{B \text{ \underline{and} } fS1(P)\} \text{ \underline{or} } \{\text{\underline{non} } B \text{ \underline{and} } fS2(P)\}$$
$$\Rightarrow \{B \text{ \underline{and} } fS1'(P)\} \text{ \underline{or} } \{\text{\underline{non} } B \text{ \underline{and} } fS2(P)\} = fS'(P) \quad . \qquad \text{QED.}$$

Binary selection, case 2, can be left to the industrious reader.

Let us now consider a predicate transformer G constructed, by means of concatenation and selection, out of a number of healthy predicate transformers, among which is fH. (This latter predicate transformer may be used "more than once": then G corresponds to a program text in which the corresponding statement H occurs more than once.) We wish to regard this predicate transformer as a function of fH and indicate that by writing G(fH), i.e. G derives, by concatenation and/or selection with other, in this connection fixed predicate transformers, a new predicate transformer. We now consider the recurrence relation

$$fH_i = G(fH_{i-1}) \tag{1}$$

which is a tractable thing in the sense that if fH_0 is as strong (weak) as fH_1, it follows via mathematical induction from the Theorem of Monotonicity that fH_i is as strong (weak) as fH_{i+1} for all i. We should like to start the recurrence relation with a constant transformer fH_0 that is either as strong or as weak as any other. We can do this for a predicate transformer as strong as any other by choosing $fH_0 = fSTOP$, given by

$$fSTOP(P) = F \quad \text{for any } P \quad .$$

(The predicate transformer fSTOP satisfies all the requirements for healthiness.) And so we find ourselves considering the sequence of predicate transformers given by $fH_0 = fSTOP$

and for $i > 0$: $fH_i = G(fH_{i-1})$ (2)

with the property that

1) all fH_i are healthy (by induction)

2) for $i \leq j$ and any P: $fH_i(P) \Rightarrow fH_j(P)$.

Because all fH_i are healthy and any $P \Rightarrow T$, we also know that for any P

$$fH_i(P) \Rightarrow fH_i(T) \quad .$$

We now recall that we were looking for fS_i such that for any P and Q and $i \neq j$ we would have $fS_i(P)$ <u>and</u> $fS_j(Q) = F$.

We can derive such predicate transformers from the fH_i. As each $fH_i(P)$ implies for the same P the next one in the sequence, we could try for $i > 0$

$$fS_i(P) = fH_i(P) \text{ \underline{and} \underline{non} } fH_{i-1}(P) \qquad (3)$$

i.e. the $fS_i(P)$ is the "incremental tolerance", but -both on account of the conjunction and on account of the negation- it is not immediately obvious that such a construction is a healthy predicate transformer. Therefore we proceed a little bit more carefully, first deriving a few other theorems about two predicate transformers fS and fS', such that fS is as strong as fS', i.e. $fS(P) \Rightarrow fS'(P)$ for any P. Another way of writing this same implication is

$$fS'(P) = fS(P) \text{ \underline{or} } \{fS'(P) \text{ \underline{and} \underline{non} } fS(P)\} \quad .$$

Referring to property 6' of healthy predicate transformers we can replace "<u>non</u> fS(P)" and find

$$fS'(P) = fS(P) \text{ \underline{or} } \{fS'(P) \text{ \underline{and} } \{fS(\underline{\text{non}}\ P) \text{ \underline{or} \underline{non} } fS(T)\}\} \quad .$$

Because $fS(\underline{\text{non}}\ P) \Rightarrow fS'(\underline{\text{non}}\ P) \Rightarrow \underline{\text{non}}\ fS'(P)$, this reduces to

$$fS'(P) = fS(P) \text{ \underline{or} } \{fS'(P) \text{ \underline{and} \underline{non} } fS(T)\} \qquad (4)$$

from which we derive (by taking the conjunction with fS(T))

$$fS'(P) \text{ \underline{and} } fS(T) = fS(P) \qquad (5)$$

and (by taking the conjunction with <u>non</u> fS(P))

$$fS'(P) \text{ \underline{and} \underline{non} } fS(P) = fS'(P) \text{ \underline{and} \underline{non} } fS(T) \quad . \qquad (6)$$

From (6) we conclude, because $fH_{i-1}(P) \Rightarrow fH_i(P)$, that our tentative definition (3) leads to

$$fS_i(P) = fH_i(P) \ \underline{\text{and}}\ \underline{\text{non}}\ fH_{i-1}(P)$$
$$= fH_i(P) \ \underline{\text{and}}\ \underline{\text{non}}\ fH_{i-1}(T) \qquad (7)$$

and because "$\underline{\text{non}}\ fH_{i-1}(T)$" is a predicate independent of P, the fS_i as defined by (3) are healthy.

Defining

$$K_0 = F \quad \text{and for } i > 0\text{:} \quad K_i = fS_i(T)$$

it is easy to show that

$$i \neq j \ \text{ implies } \ K_i \ \underline{\text{and}}\ K_j = F \qquad (8)$$

This is proved by a reductio ad absurdum. Let $i < j$ and suppose $K_i \ \underline{\text{and}}\ K_j \neq F$; then there exists a point v in state space such that

$$K_i(v) \ \underline{\text{and}}\ K_j(v) = \underline{\text{true}} \quad .$$

However, $K_i(v)$ implies $fH_i(T)(v)$ which implies $fH_{j-1}(T)(v)$ -because $j-1 \geq i$ - which implies $K_j(v) = \underline{\text{false}}$ and this is the contradiction we were after. In other words: in each point in state space at most one K_i is $\underline{\text{true}}$.

From (7) combined with $fH_i(P) \Rightarrow fH_i(T)$ it follows that

$$fS_i(P) = K_i \ \underline{\text{and}}\ fH_i(P) \qquad (9)$$

which together with (8) leads to the conclusion that for any P and Q

$$i \neq j \ \text{ implies } \ fS_i(P) \ \underline{\text{and}}\ fS_j(Q) = F \qquad (10)$$

and this is exactly the relation we have been looking for.

In passing we note that, on account of (9), $K_i = F$ implies $fS_i(P) = F$; on account of (7) this tells us that for any P $fH_i(P) \Rightarrow fH_{i-1}(P)$; we also know that $fH_{i-1}(P) \Rightarrow fH_i(P)$ for any P and we conclude $fH_i(P) = fH_{i-1}(P)$. As this holds for any P, we conclude $fH_i = fH_{i-1}$ and therefore

$$fH_{i+1} = G(fH_i) = G(fH_{i-1}) = fH_i \quad .$$

In other words

$$K_i = F \text{ implies } \ fH_j = fH_{i-1} \quad \text{for } j \geq i$$
$$\text{and } \ K_j = F \qquad \text{for } j > i \qquad (11)$$

Returning to (10) we conclude that with the aid of our sequence fS_i we can now form two new healthy predicate transformers, firstly

$$fH(P) = (\underline{A}\ i{:}\ 1 \leq i{:}\ fS_i(P)) \qquad ,$$

but that one, although healthy, is not interesting because on account of (10) it is identically F; and secondly

$$fH(P) = (\underline{E}\ i{:}\ 1 \leq i{:}\ fS_i(P)) \qquad , \qquad (12)$$

The latter one is not identically F and we call it a predicate transformer "composed by recursion". In formula (12), for each point v in state space, such that $fH(P)(v) = \underline{\text{true}}$, the existential quantifier singles out a unique value of i.

Alternatively we may write

$$fH(P) = (\underline{E}\ j{:}\ 1 \leq j{:}\ fH_j(P)) \qquad . \qquad (13)$$

It is by now most urgent that we relate the above to our intuitive understanding of the recursive procedure: then all our formulae become quite obvious.

First a remark about the Theorem of Monotonicity: it just states that if we replace a component of a structure by a more powerful one, the modified structure will be at least as powerful as the original one. (Consider, for instance, an implementation of a programming language that leads to program abortion when integer overflow occurs, i.e. when an integer value outside the range [-M, +M] is generated. When we modify the machine by increasing M, all computations that were originally feasible, remain so, but possibly we can do more.)

Now for the recursion. All we have been talking about is a recursive procedure (without local variables and without parameters) that could have been declared by a text of the form

$\underline{\text{proc}}$ H: ..... H ..... H ..... H ..... $\underline{\text{corp}}$

i.e. a procedure H that may call itself from various places in its body. Mentally we are considering a sequence of procedures H_i with

$\underline{\text{proc}}$ H_0: STOP $\underline{\text{corp}}$

$\underline{\text{proc}}$ H_i: ..... H_{i-1} ..... H_{i-1} ..... H_{i-1} ..... $\underline{\text{corp}}$

Our rules

$$fH_0 = fSTOP \quad \text{and for } i > 0 \quad fH_i = G(fH_{i-1})$$

are such that the predicate transformer fH_i corresponds to our intuitive understanding of the call of procedure H_i. In terms of the procedure H, fH_i describes what a call of the procedure H can do under the additional constraint that the dynamic recursion depth will not exceed i. In particular, $fH_i(T)$ characterizes the initial states such that the procedure call will terminate with a dynamic recursion depth not exceeding i, while K_i characterizes those initial states such that a call of H will give rise to a maximum recursion depth exactly = i. This intuitive interpretation makes our earlier formulae quite obvious, fH(T) is the weakest pre-condition that the call will lead to a terminating computation.

The Theorem of Monotonicity was proved for predicate transformers formed by concatenation and/or selection. If in the body of H one of the predicate transformers fS is replaced by fS', as weak (strong) as fS, then G'(fH) will be as weak (strong) as G(fH), giving rise to an fH'_i as weak (strong) as fH_i; as a result the Theorem of Monotonicity holds also for predicate transformers constructed via recursion.

Our axiomatic definition of the semantics of a recursive procedure

$$\begin{array}{ll} & fH_0 = fSTOP \text{ and} \\ \text{for } i > 0: & fH_i = G(fH_{i-1}) \text{ and} \\ \text{finally:} & fH(P) = (\underline{E}\ i: i > 0: fH_i(P)) \end{array}$$

is nice and compact, in actual practice it has one tremendous disadvantage: for all but the simplest bodies, it is impossible to use it directly. $fH_1(P)$ becomes a line, $fH_2(P)$ becomes a page, etc. and this circumstance makes it often very unattractive to use it directly. We cannot blame our axiomatic definition of the recursive procedure for this unattractive state of affairs: recursion is such a powerful technique for the construction of new predicate transformers that we can hardly expect a recursive procedure "chosen at random" to turn out to be a mathematically manageable object. So we had better discover which recursive procedures can be managed intellectually and how. This is nothing more nor less than asking for useful theorems about the semantics of recursive procedures.

* * *

Now we are going to prove the Fundamental Invariance Theorem for Recursive Procedures.

Consider a text, called H", of the form

H": ..... H' ..... H' ..... H' .....

to which corresponds a predicate transformer fH", such that for a specific pair of predicates Q and R, the assumption $Q \Rightarrow fH'(R)$ is a sufficient assumption about fH' for proving $Q \Rightarrow fH''(R)$. In that case, the recursive procedure H given by

<u>proc</u> H: ..... H ..... H ..... H ..... <u>corp</u>

(where we get this text by removing the dashes and enclosing the resulting text between the brackets <u>proc</u> and <u>corp</u>) enjoys the property that

$$\{Q \text{ \underline{and} } fH(T)\} \Rightarrow fH(R) \qquad (14)$$

(The tentative conclusion $Q \Rightarrow fH(R)$ is wrong as is shown by the example <u>proc</u> H: H <u>corp</u> .)

We show this by showing that then for all $i \geq 0$

$$\{ Q \text{ \underline{and} } fH_i(T)\} \Rightarrow fH_i(R) \qquad (15)$$

and from (15), (14) follows trivially. Relation (15) holds for $i = 0$, and we shall show if it holds for $i = j-1$, it will hold for $i = j$ as well.

In the formulation of the Fundamental Invariance Theorem for Recursive Procedures we have mentioned "a pair of predicates Q and R"; we did so, because besides the co-ordinate variables of the state space, in which the computations evolve, and the constants, they may contain free variables as well and they are paired by the fact that they are the same in a pair Q and R. For instance, both Q and R may end with "<u>and</u> $(x = x_0)$", where "x" is a co-ordinate variable and "x_0" a free variable, thus expressing that the value of x will remain unchanged, whatever its initial value. To denote a specific set (or sets) of free variable values, we shall use small letters, supplied as subscripts. Our statement of affairs, say

$$Q \Rightarrow fH''(R)$$

is then written down more explicitly as

$$Q_e \Rightarrow fH''(R_e)$$

in order to indicate that Q and R are coupled by a set of free variables. (As subscripts I shall use "e" for external and "i" for internal.)

Let us first consider, for the sake of simplicity, the case that the text H" contains a single reference to H'. In the evaluation of $fH''(R_e)$, let $P1_e$ be the argument that, working backwards, is supplied to fH'; with

$$P2_e = fH'(P1_e)$$

we can then write $$fH''(R_e) = E(P2_e) \qquad (16)$$

We can regard E as a predicate transformer operating on its argument $P2_e$, but considered as predicate transformer it is not necessarily healthy: it may violate the Law of the Excluded Miracle. It enjoys, however the other three properties:

$$P = Q \text{ implies } E(P) = E(Q)$$
$$E(P \underline{\text{ and }} Q) = E(P) \underline{\text{ and }} E(Q)$$
$$E(P \underline{\text{ or }} Q) = E(P) \underline{\text{ or }} E(Q)$$

and therefore also the fifth:

$$P \Rightarrow Q \text{ implies } E(P) \Rightarrow E(Q) \quad .$$

The statement that with regard to the predicate pair Q and R the assumption $Q \Rightarrow fH'(R)$ is a sufficient assumption about fH' in order to prove $Q \Rightarrow fH''(R)$ amounts more explicitly to the following statement:

There exist for the free variables occurring in Q and R a set i of values (in general functionally dependent on the set e), such that

$$R_i \Rightarrow P1_e$$
$$Q_e \Rightarrow E(Q_i) \quad . \qquad (17)$$

(For instance, consider the statement

H": n:= n - 1; H'; n:= n + 1

with Q and R both: $n = n_0$, where n_0 is a free variable. Our proof for

$$(n = n_e) \Rightarrow fH''(n = n_e)$$

can be based on the assumption

$$(n = n_i) \Rightarrow fH'(n = n_i)$$

with $\quad n_i = n_e - 1$.
Here R_e and Q_e are both: $n = n_e$ and R_i and Q_i are both: $n = n_i$.)

When we are now able to show that

$$fH_j(T) \Rightarrow E(fH_{j-1}(T)) \qquad (18)$$

then it follows from (17) that

$$\{Q_e \text{ \underline{and} } fH_j(T)\} \Rightarrow E(Q_i \text{ \underline{and} } fH_{j-1}(T))$$

and as a result $\{Q_i$ _and_ $fH_{j-1}(T)\} \Rightarrow fH'(R_i)$ is then a sufficient assumption about fH' to conclude that $\{Q_e$ _and_ $fH_j(T)\} \Rightarrow fH''(R_e)$. As fH_j depends on fH_{j-1} as H" on H', this would conclude the induction step and (14) would have been proved.

We have two holes to fill: we have to show (18) and we have to extend the line of reasoning to texts of H", containing more than one reference to H'. Let us first concentrate on (18).

We have defined $fH_j = G(fH_{j-1})$, but because for any P, we have $fH_{j-1}(P) \Rightarrow fH_{j-1}(T)$, an identical definition would have been

$$fH_j = G(fH_{j-1}(T) \text{ \underline{and} } fH_{j-1})$$

i.e. each predicate formed by applying fH_{j-1} is replaced by its conjunction with $fH_{j-1}(T)$. And therefore, instead of

$P1 = fS(T)$ (i.e. P1 is the argument supplied to fH' in the evaluation of fH"(T).)
$P2 = fH_{j-1}(P1)$
$fH_j(T) = E(P2)$

we could have written equally well

$P1 = fS(T)$
$P2 = fH_{j-1}(P1)$
$fH_j(T) = E(P2 \text{ \underline{and} } fH_{j-1}(T))$.

But $\{P2$ _and_ $fH_{j-1}(T)\} \Rightarrow fH_{j-1}(T)$ and therefore, because the transformer E enjoys the fifth property, we are entitled to conclude

$$fH_j(T) \Rightarrow E(fH_{j-1}(T)) \qquad \text{i.e. relation (18) .}$$

To fill the second hole, viz. that in the text called H" more than one reference to H' may occur, is easier. Working backwards in the evaluation of $fH''(R_e)$ means that we first encounter the innermost evaluation(s) of fH', whose argument does not contain fH'. For those predicate transformers we apply our previous argument, showing that for them the weaker assumption Q <u>and</u> $fH_{j-1}(T) \Rightarrow fH'(R)$ is sufficient. Then its value is replaced by Q_i (or Q_{i1} if you prefer) and we start afresh. In this way the sufficiency of the weaker assumption about fH' can be established for all occurrences of fH' -only a finite number!- in turn.

* * *

For the recursive routines of the particularly simple form

<u>proc</u> H: <u>if</u> B <u>then</u> S1; H <u>else</u> <u>fi</u> <u>corp</u>

we can ask ourselves what must be known about B and S1, when we take for R the special form Q <u>and</u> <u>non</u> B. Then

$$fH''(Q \text{ and non } B) = \{B \text{ and } fS1(fH'(Q \text{ and non } B))\} \text{ or } \{Q \text{ and non } B\} .$$

In order to be able to conclude $Q \Rightarrow fH''(Q \text{ and non } B)$ on account of $Q \Rightarrow fH'(Q \text{ and non } B)$, the necessary and sufficient assumption about fS1 is

$$\{Q \text{ and } B\} \Rightarrow fS1(Q) .$$

Procedures of this simple form are such useful elements that it is generally felt justified to introduce a specific notation for it, in which the recursive procedure remains anonymous: it should contain as "parameters" the B and the S1 and we usually write

<u>while</u> B <u>do</u> S1 <u>od</u> .

With the statement S of the above form, we have now proved that

$$\{Q \text{ and } B\} \Rightarrow fS1(Q) \quad \text{implies} \quad \{Q \text{ and } fS(T)\} \Rightarrow fS(Q \text{ and non } B)$$

This is called "The Fundamental Invariance Theorem for Repetition".

Acknowledgements.

Acknowledgements are due to the members of IFIP Working Group 2.3 on Programming Methodology with whom I had the privilege to discuss a preliminary version of this paper at the Munich meeting in April 1973. Among them, special thanks deserve M.Woodger, whose inspiring influence and assistance with respect to this work extended over the weeks both before and after that meeting, J.C.Reynolds, who has drawn my attention to an incompleteness in my first proof of the Invariance Theorem and finally, of course, C.A.R.Hoare, because without his innovating work mine would not have been possible at all.

Special thanks are further due to C.S.Scholten for cleaning up several of my formal proofs and to my collaborators at the Technological University, Eindhoven, W.H.J.Feijen and M.Rem for their encouragement and comments while the work was done.

Eindhoven, 8th May 1973