<u>Paying logical conscience-money to the fair daemon</u>.

The traditional formulation of the difference between total and partial correctness is: in the case of total correctness we prove that a correct result will be produced, in the case of partial correctness we prove that no incorrect result will be produced, or. more explicitly, in the case of partial correctness we prove that either a correct result will be produced or that the mechanism fails to terminate. As a result, a proof of partial correctness supplemented with a proof of termination yields a proof of total correctness.

In the sixties R.W.Floyd did just that, basing himself on a flowchart language. A year later, C.A.R.Haore gave semantic axioms and proof rules for partial correctness for a language with more syntactical structure. Floyd's approach was still very operational, and he seemed to consider the semantics of his program --as a "working mechanism"-- also defined in the case of non-termination. Hoare, who only concerned himself with partial correctness, did not need to talk about termination and his approach is in that sense less operational, less "mechanical".

I liked Hoare's approach --it was at the time that I was coining the term structured programming-- much better than Floyd's, but was also dissatified by it, because it was clearly incomplete: his axoims and proof rules permitted one to translate "<u>while</u> B <u>do</u> S <u>od</u>" into "<u>while</u> B <u>do</u> skip <u>od</u>" .

In the early seventies I developed the predicate transformers as a means for defining program semantics. My approach differed from the previous ones in two main aspects: I restricted myself to total correctness and allowed non-determinacy.

The operational interpretation of my formal system is that for those initial states in which termination is not guaranteed, I have left the semantics undefined. The inverse interpretation is that my formal semantics for a program S are only concerned with those initial states satisfying wp(S, T) --that "the answer" is only defined as a partial "function"-- and that for all initial states <u>not</u> satisfying wp(S, T) any implementation is totally free to choose its reaction: the mechanism may embark upon an infinite computation, it may even evaporate.

That the formalism is restricted to what can be accomplished by terminating computations has the very great advantage that I really don't need to talk about non-terminating ones. This is a great advantage, because the question of terminatio nontermination is always couched in operational terminology, from which I could now depart: if so desired I can ignore the circumstance that my text also permits the interpretation of executable code.

I had to pay a price for that puxury. For the program

```
S:        do x > 0 → x:= x + 1
           [] x > 0 → x:= 0
          od
```

I have only defined that for $x \leq 0$, S is equivalent to a skip, for $x > 0$ I have not only defined nothing --that is not too bad, because: termination is not guaranteed, isn't it?-- but have not even the tools for defining that, if it terminates, it will terminate with $x = 0$. The purpose of this note is to show how I can attach a meaning to such a program S by regarding it as an abbreviation of a program S' .

Consider the general repetitive construct

```
DO:       do B1 → S1 [] ... [] Bn → Sn od
IF:       if B1 → S1 [] ... [] Bn → Sn od
BB:       B1 or ... or Bn
```

and suppose that we have proved for a certain P :

$$(P \text{ and } BB) \Rightarrow wp(IF, P) \qquad (1)$$

With a ghost variable t we derive the primed system:

Bj' = Bj and $t > 0$ (hence BB' = BB and $t > 0$)

Sj' = Sj; t:= t - 1

P' = P and $t \geq 0$.

Then (1) implies (P' and BB') ⇒ wp(IF', P')

furthermore (P' and BB') ⇒ wdec(IF', t) and $t > 0$.

Hence our well-known theorem about the repetitive construct allows us to conclude

P' ⇒ wp(DO', P' and non BB') .

Expressed in the old P and BB , this post-condition reduces to

$$P \ \underline{\text{and}}\ t \geq 0 \ \underline{\text{and}}\ (\underline{\text{non}}\ BB \ \underline{\text{or}}\ t \leq 0)$$

which implies $$t > 0 \ \Rightarrow\ (P \ \underline{\text{and}}\ \underline{\text{non}}\ BB) \qquad (2)$$

Operationally, (2) can be interpreted as "when S stops, we could have chosen for t such a large initial value that its final value is still positive, hence P and non BB has been established". By introducing the ghost variable t which --in the operational sense-- does not only count the repetitions but also forces termination, we have related S to a program S' for which I can prove total correctness in my usual manner.

* * *

The above formal manipulations are not in any sense deep. But I am very pleased. Instead of building my theory upon mechanisms which may terminate or not, I build my theory on texts to which --but I don't need to remember that-- terminating mechanisms can be made to correspond. By abbreviating some texts (from S' to S) I indicate a reduction of the mechanism, and the reduced mechanism may fail to terminate....

As said, I am very pleased, for I hope that this "trick" will enable us to cope in multiprogramming without mathematical problems with the kind of "bounded but unspecified" nondeterminacy that we have captured in the up till now intractable metaphor of "a fair daemon".

10th of January 1977

Plataanstraat 5
NL-4565 NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow