I recently obtained a tattered, partially destroyed copy of a manuscript that appears to be an early draft of EWD1013. This raised a difficult question for me. Should I honor Dijkstra's right as a scientist to be judged only by what he publishes and destroy the manuscript, or should I share with others this rare chance to observe the evolution of his ideas? In the end, I felt that the insight afforded by the manuscript was great, and Dijkstra's reputation is too secure to be damaged by this disclosure. I have therefore attached a copy of the manuscript, which I took the liberty of having retyped.

Leslie Lamport

EWD1013 - 0 (draft)

Position paper on "termination"

Life is a very complicated business if you want to do it well. This is because anything of any importance is always a many-sided affair and none of its different aspects may be neglected, while at the same time, in order to do
the whole job well, the different concerns have to be separated as ruthlessly as possible.

Before embarking on a major research topic, you had better choose your

[this portion of the manuscript is illegible - L.L.]

about your "how" you will almost certainly discover that in major parts of your investigations automatic computers, with all their quirks and physical limitations, are totally irrelevant and had better be forgotten.

* *
*

One area in which it has proved to be very fruitful to forget that automatic computers exist is programming. One forgets that computers exist, one ignores that one's programs admit --in another world, so to speak-- the interpretation of executable code and treats the program text as a mathematical object in its own right. All by itself it is not a very interesting object, but in combination with its functional specification, the statement that the program meets its functional specification is a theorem. At that intellectual level, programming is about how to design such theorems and their proofs. And from sad experience we all know that this activity reveals a core challenge, if not the core challenge, of computing science, viz. "How not to make a mess of it and how not to get confused in the complexities of one's one making.".

What is at that level the role of the programming language used? Essentially only one, viz. to define the proof obligations engendered by presenting the combination of program and functional specification as a theorem.

In this part of the exercise it is clearly totally irrelevant whether the programming language used to express this theorem has been implemented. It is even irrelevant whether an economically acceptable implementation is technically feasible.

[the top of this page was missing from the manuscript - L.L.]

machine-- computations meeting the specification. In this sense, a programming language emerges as a contract between programmer and the implementer, stating the rights and the obligations for both partners in the deal. If the programmer has met his proof obligation, he is entitled to the correct results; the implementer has the obligation to see to it that the correct result is produced, but has the right to refuse programs violating the stated constraints.

Let us now consider the little program fragment

```
print("done")
```

While still dealing with abstract programs I am perfectly willing to accept this as a program that cannot terminate without printing the string "done". I am even willing to accept this as a program that under the assumption of a sufficiently benevolent scheduler will, sooner or later, terminate. Such "behaviour" is thinkable and at that level thinkability is the only thing that matters. (I may add that my willingness has been a very active one. Inclusion of termination amounts to the inability of regarding a program purely in terms of Hoare triples; how to go beyond Hoare's concept of pre- and postconditions has been one of my earlier contributions.)

So far, so good. But things change drastically as soon as we start talking about an implemented programming language. Then the programming language emerges as a contract stating rights and obligations, and there are such things as void obligations.

I call an obligation void if it is impssible to detect if it has not been fullfilled. I can easily promise to think at least three times per week about you, but that is a very cheap promise because no one will ever be able to show that I failed to fulfill my commitment. As a promise it is void.

Assume now that the language definition is such that above programming fragment is eventually to print "done" and then terminate. This would be the prototype of a void obligation for the implementer. Firstly, nothing prevents him from implementing it in a way semantically equivalent to

```
  i := 100000
; do i > 0 -- i := i-1 od
; print("done")
```

As user of his sytem you may be disappointed that takes so long to print the string "done" and terminate, but the implementer can shrug his shoulders and say "Your computer must have been running unusually slowly at the time! Try again.". What you may consider as a regrettable breach of contract on his part won't cause the implementer a single sleepless night because he knows that though his obligation was void, he has fulfilled it. After all, the program did eventually terminate.

Secondly, suppose that you pester him and start threatening with a law suit if you don't get a better implementation. This time the implementer agrees to replace his previous implementation, and now he implements the fragment semantically equivalently to

```
  do true -- skip od
; print("done")
```

And now we are in the paradoxical situation that the implementer knows that he has violated the contract, for he knows that his product will not terminate. At the same time he knows that, no matter how many experiments you take, no matter how many instances of his program you start, you will never be able to produce the evidence that he has violated the contract. So, again you won't cause him a single sleepless night.

The moral of the story is that void obligations should not occur in contracts.

Finally I would like to point out that in the case of termination the implementor's situation is drastically different from that of the quality

[Some dozen lines of the manuscript were here blotted out by a stain. A preliminary chemical analysis revealed the presence of barley, possibly accompanied by a trace of hops. - L.L.]

settled by the roulette in question.

But termination is not a probabilistic notion and if the implementer is sued, he will be acquitted for lack of evidence. My conclusion from the above is that termination, being an unworkable notion, can be ignored with impunity.

prof. dr. Edsger W. Dijkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188

Austin, 13 October 1987