Position paper on "fairness"

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 target very carefully because the borderline between the insipid and the impossible is very thin. In other words, you had better be fully aware why you engage yourself on the project. At the same time you had better separate that concern entirely from the technical question of how to achieve your goal. The why and the how are completely distinct concerns, and nothing is gained by mixing the two. On the contrary.

Let us accept as Thesis that the research of a Computing Scientist is eventually concerned with automatic computing. That leaves a wide range: from improving the equipment to improving our abilities of using the equipment. In one way or another, automatic computers and their usage provide the soil in which our "why" has its roots. Fine.

But in order to do your research successfully, you also have to think very carefully about how to conduct your research, and this time because the borderline between success and failure is preciously thin. And in carefully thinking 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 funtional 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 ~~whether~~ an economically acceptable implementation is technically feasible.

Let me elaborate for a short while. I have to respect the strictly limited size of my head and can deal with only one thing at a time. This means that in designing my program the fixing of certain details will have to be postponed; it also means that meeting certain proof obligations will be postponed. Such "abstract" programs often distinguish themselves by the inclusion of some powerful statements that almost certainly defy automatic implementation, and they are usually grossly nondeterministic. I may, for instance, meet a proof obligation whose operational interpretation is that the nondeterminacy gives so much freedom that there exists a scheduling that does not lead to deadlock. That at the same time an implementation that would avoid deadlock defies imagination should not deter me at all. Implementation is only an operational concern that plays no role in the theorem I am developing.

A totally new set of considerations enters the picture by the time that I would like to use my theorem in the sense that I would like to have my program executed because I am interested in the result of the ensuing computations. For that purpose the programming language I wrote the program in has to be implemented on a sufficiently powerful machine. And suddenly the language definition appears in a totally different light. It gives the implementer rights and obligations; he has the right to refuse successful completion of the execution of programs in which syntactic or semantic constraints have been violated, he has the obligation to generate --within the capacity of the machine-- computations meeting the specification. In this sense, a programming language emerges as a contract between programmer and implementer, stating the rights and the obligations for both partners in the deal. If the programmer has met his proof obligations, 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

```
  b:= true
; do b -- print(0)
  [] b -- print(1); b:= false
  od                                .
```

While still dealing with abstract programs I am perfectly willing to accept this as a program that prints an arbitrary number of zeros and stops after the printing of the first one. I am even willing to accept this as a program that under the assumption of a sufficiently benevolent scheduler will, sooner or later, print that one and then stop. 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 unbounded nondeterminacy amounts to the loss of or-continuity; how to design proofs that do not rely on or-continuity has been one of my more recent 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 impossible to detect if it has not been fulfilled. 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 fulfil my commitment. As a promise it is void.

Assume now that the language definition is such that above programming fragment is to print a finite string of zeros of arbitrary length followed by a one. 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

```
  b:= true
; do b -- print(1); b:= false od     .
```

As user of his system you may be dissappointed that it never prints a zero, but the implementer can shrug his shoulders and say "You must have had bad luck! 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, zero is one of the arbitrary numbers.

Secondly, suppose that you pester him and start threatening with a law suit if you don't see any zeros printed. This time the implementer agrees to change the scheduler, and now he implements the fragment semantically equivalently to

```
  b:= true
; do b -- print(0) od     .
```

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 never print a one. At the same time he knows that, no matter how many experiments you take, now 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 fairness the implementor's situation is drastically different from that of the quality controller in a roulette factory. There is no experiment that he can take to assure that a roulette coming out of the factory is unbiased. With a perfectly fair roulette it is possible that each time he turns the roulette and throws the ball, the ball will end up at zero. Unlikely, but perfectly possible. Therefore the quality assurance man from the roulette factory will never tell you that a roulette is unbiased: his experiments will tell him the probability that it is unbiased and if you want that probability to be very high, you will have to pay more because he has to take a longer series of experiments. The dissatisfied customer can still try to sue the roulette manufacturer, not because he has been cheated by the latter but only on account of the very high probability that he has done so. In a case like that, the conscientious judge can give the roulette manufacturer only a probabilistic fine; whether or not it should be paid can be settled by the roulette in question.

But fairness 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 fairness, being an unworkable notion, can be ignored with impunity.

Austin, 14 October 1987

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