Why correctness must be a mathematical concern.

(Inaugural lecture for the "Chaire Internationale d'Informatique" at the Université de Liège, Belgium.)

Ladies and Gentlemen:

The topic to which this new international chair at Liège has been devoted has many names. On the continent of Europe the recently coined name "Informatics" has become generally accepted; in the Anglo-Saxon world the much older term "Computer Science" is most commonly used, though occasionally replaced by the now more appropriate term "Computing Science". The latter term is more appropriate because it expresses quite clearly that not a piece of equipment, but an activity constitutes the core of its subject matter. By its very nature, this subject matter is highly technical; an inaugural address, however, is specifically the occasion at which technicalities should be avoided, and it is my purpose today to explain to you the significance of the topic without explaining in any detail the topic itself.

You cannot expect me to explain in a few words what mathematics is all about: in order to grasp that, one has to do mathematics for many years oneself. But I would like to show you one simple argument in order to give you in a nutshell some of the flavour of mathematics.

Consider the following silly game to be played by a single person with an urn and as many white balls and black balls as he needs. To begin with an arbitrary positive number of balls is put into the urn, and as long as the urn contains two or more balls, the player repeats the following move: he shakes the urn and, without looking, he takes two balls from the urn; if those two balls have the same colour he throws one black ball back into the urn, otherwise he returns one white ball into the urn. Because each move decreases the total number of balls in the urn by 1 , the game is guaranteed to terminate after a finite number of moves, and it is not difficult to see that the game ends with exactly 1 ball in the urn. The question is: "What can we say about the colour of that final ball when we are given the initial contents of the urn?".

Well, we can try all possible games! The games that start with 1 ball in the urn are very simple. Because they involve no move at all, we might call them "the empty games", and could represent them as

○

and

●

respectively.

The games of 1 move are not very complicated either: we can represent them by

○ ○ ⟶ ●

○ ● ⟶ ○

● ● ⟶ ●

respectively.

But among games of 2 moves, life becomes already more complicated. We might represent them as follows (note that there are six possible games):

$$\begin{tikzcd}
\circ\ \circ\ \circ \arrow[r] & \circ\ \bullet \arrow[r] & \circ \\
 & \circ\ \circ \arrow[r] & \bullet \\
\circ\ \circ\ \bullet \arrow[ur, "\circ\bullet"] \arrow[r, "\circ\circ"'] & \bullet\ \bullet \arrow[r] & \bullet \\
 & \circ\ \bullet \arrow[r] & \circ \\
\circ\ \bullet\ \bullet \arrow[ur, "\bullet\bullet"] \arrow[r, "\circ\bullet"'] & \circ\ \bullet \arrow[r] & \circ \\
\bullet\ \bullet\ \bullet \arrow[r] & \bullet\ \bullet \arrow[r] & \bullet
\end{tikzcd}$$

It is clear that such a tabulation becomes extremely tedious. Besides that, one is never ready with such a tabulation: suppose that we have made the tabulation of games up to 100 moves, then we know nothing to say when faced with an urn that initially contains 300 balls.

Looking at the three single moves possible, we observe that the last two (○● ⟶ ○ and ●● ⟶ ●) leave the number of white balls in the urn unchanged, while the first move (○○ ⟶ ●) reduces the number of white balls in the urn by 2. In other words, each move leaves the so-called "parity" of the number of white balls in the urn unchanged: an even number of white balls in the urn remains even, and an odd number of white balls in the urn remains odd. In short: if the initial number of white balls is even, the final ball is black, and if the initial number of white balls is odd, the final ball is white. And that answers the question!

Note that this single argument settles the question for <u>all</u> initial contents of the urn, and per initial contents for <u>all</u> of the perhaps many possible games.

The above is a good carrier for some of the flavour of mathematics:
a) the answer to our question was a very general one in the sense that it is pertinent to an unbounded number of different cases (here: possible games);
b) the answer to our question was very precise; this in contrast to the soft sciences, such as sociology, in which generality is traditionally achieved by vagueness, i.e. precision's opposite;
c) the answer to our question has been justified by a convincing argument, i.e. the type of proof that makes mathematical statements more trustworthy than anything else we utter.

In a very elementary, but fundamental sense the example is also typical for some of the mathematical arguments that are relevant to automatic computing. Replace the initial contents of the urn by "the input", the rules of the game by "the program", the game as it evolves by "the computation", and the colour of the final ball by "the result", and the above example gives you in a nutshell the bare structure of one of the most effective ways we know of reasoning about programs. (Thanks to the fact that in each move the player draws two arbitrary balls, our example even reflects the characteristics of so-called "non-deterministic algorithms", in which the computational history need not be uniquely determined by the input that has been supplied explicitly.)

A while ago I mentioned as a leading characteristic of mathematical statements their "generality" in the sense that they are applicable to a very large, often even unbounded number of cases. And it is good thing to realize that almost all computer applications are "general" in that very same sense. In a banking application one does not design a system that can <u>only</u> transfer $ 100 from the account of a certain mr.Smith to that of a certain mr.Brown!

On the contrary, the banking system should be able to cope with the transfer of almost <u>any</u> amount from <u>any</u> account to <u>any</u> <u>other</u> account. So much for the generality.

The next characteristic of mathematics that I mentioned was precision. And it is a good thing to realize that almost all computer applications to be worthwhile have to have that characteristic of precision as well. Again the banking application may serve as an illustration. It is not only that all accounts are kept track of accurately to the dollarcent, it is much more: for it to be useful, the system as a whole must have a number of very precise properties. As far as the internal transfers from one account to another are concerned, the system must reflect the Law of Conservation of Money: the sum of the accounts must remain constant because each deduction from one account must be compensated by an equal increase of another account.

The last characteristic I mentioned was trustworthiness. And it is a good thing to realize that almost all computer applications to be worhtwhile must be trustworthy. What is the purpose of designing a banking system with the best intentions, when in actual practice it fails to keep correctly track of the flow of money? Not only does it have to do so correctly, but before installing it and switching over to it we must have solid grounds for believing that it will do so correctly. Installing the system without such solid grounds would, in view of the risks involved, be an act of sheer irresponsibility.

Having thus shown that each worthwhile computer application shares with mathematics the latter's three leading characteristics, I hope to have convinced you that by its very nature responsible systems design and development --in short: programming reliably-- <u>must</u> be an activity of an undeniably mathematical nature. And having chosen the banking application as illustration, I hope to have convinced you that this conclusion is not confined to the application of computers to science and technology.

Though the conclusion seems unescapable, I should mention for the sake of completeness that not everybody is willing to draw it. You see, mathematics is about thinking, and doing mathematics is always trying to think as well as possible. By an unfortunate accident of history, however, programming became an industrial activity in the United States at a moment that the American manager was extremely fearful of relying on the education and the intelligence of his company's employees. And management tried to organize the industrial programming task in such a way that each individual programmer would need to think as little as possible. This has been a sad mistake, but the more management, with this conception of programming, failed to meet its deadlines and to achieve reasonable quality standards, the more has been invested in these ill-directed efforts. And now we have reached the paradoxical situation that, while the evidence for the intrinsic difficulty of the programming task has become more and more convincing each year, the recognition of the difficulty has become, both politically and socially, more and more unpalatable. It should be noted that under these circumstances foresight and courage, if not also the tenacity of a bull-terrier, are required for the foundation of an international chair of informatics at which programs and programming are considered to be worthy of our serious scientific attention.

Let me next indicate, in the broadest outline possible, what this serious scientific attention could cover. I shall keep this outline broad for two reasons: firstly this is not the occasion to bore you with technicalities, secondly I hope to give an explanation with which the subsequent occupants of this chair won't disagree too much.

It is now the time to confess that my example of the game with the urn filled with a finite number of balls, each of which is white or black, though in one respect absolutely typical, is very misleading in another respect: compared to the actual situation in programming it is such a gross oversimplification, that the use of this example in an expository lecture is almost an intellectual dishonesty.

So that you may understand how terribly gross the oversimplification really is, I would like you to realize that something as trivial as "a factor of ten" --something we usually treat as a gradual difference from which we can abstract-- makes in practice an almost essential difference. Once I was called to explain to a housewife what tremendous difference a mere factor of ten makes. I asked her, how many children she had, knowing that she had six. The lady saw the point. Compared to the programming problem, the urn example is an oversimplification that is orders of magnitude worse than merely ignoring a factor of ten.

In the urn example, the initial state is fully characterized by two integers, viz. the number of white balls and the number of black balls respectively. That specification of the input is ridiculously simple compared to, for instance, the class of possible inputs a compiler must be able to process, viz. all texts that are legal or illegal program texts in the programming language concerned.

In the urn example the final state, i.e. the colour of the final ball, represents only a single bit of information. That, again, is ridiculously simple compared to the output required from standard programs such as compilers, for which the representation of the result often requires millions of bits.

And finally, in the urn example I could state "the rules of the game" in one or two sentences, whereas in the practice of automatic computing the program --i.e. "the rules of the computational game"-- often requires many thousands, and sometimes apparently even millions of lines of text.

In the urn example, a single argument based on the rules sufficed for <u>all</u> possible games. In computing, a single argument based on the program text should analogously suffice for <u>all</u> computations that are possible under control of that program. The necessary economy of thought requires as its ultimate consequence that we learn how to reason about programs without mentioning the corresponding computational processes at all. This means no more and no less than that we must learn how to come to grips with the program text as a mathematical object in its own right. We must be able to deal with it directly, rather than via the detour of the class of all corresponding computations.

It is abundantly clear that significant progress in programming, in reasoning about programs, and in the design of programming languages will only materialize, provided we learn how to do this, while temporarily _fully_ ignoring that our program texts also admit the interpretation of executable code, because it is precisely that interpretation that clutters up our minds with all the computational processes, which truly baffle the imagination. In short: for the effective understanding of programs we must learn to abstract from the existence of computers.

This abstraction is not easy. For most people trained as electronic engineers it is even impossible --though not for all: it is a pleasure to mention Niklaus Wirth from Zurich as an exception-- . This is not surprising, for suddenly they find themselves invited to abstract from what they have been taught to regard as the core of their craft. As you might expect, vested interests inhibit a widespread recognition of the obvious conclusion, viz. that as a rule departments of electronic engineering actually deprive their graduates from the ability to understand later in life what computing science is really about.

In the ears of the traditional mathematician the suggestion to abstract from the existence of computers sounds much more natural. He should feel quite at home with the idea of regarding, reading, writing, and manipulating program texts just like any other mathematical formulae. But there is one big, big difference: never in his life he has encountered such big formulae! Remember the drastic difference already made by a single factor of ten, and realize that program texts present themselves to the traditional mathematician as formulae that are not one, but several orders of magnitude bigger than the formulae he used to deal with! In relation to our mathematical tradition, it is just a drastic problem of scale, so drastic, as a matter of fact, that quite a few mathematicians, firmly rooted in the past, are quite unable to recognize programming as a mathematical activity. Research and education in computing sciences are, however, more concerned with the mathematicians of the future.

By now you may have some feeling for the type of topics that would be fully appropriate for this international chair of informatics.

The most general topic, and also the one of the widest significance, could be called "the scaling up of mathematics". Such scaling up would imply a different style of mathematical texts in general, and the use of more appropriate notations in particular. By and large, current mathematical style has been determined by fashion, and current mathematical notation by accident.

The degree in which separable concerns are, indeed, separated is, for instance, an aspect of mathematical style, but as far as I know students of mathematics are not taught today to take in this respect a conscious decision.

Similarly, the adequacy of chosen notational conventions can be judged by their suitability to our manipulative needs. Again, as far as I know, students of mathematics are not taught today to screen notational conventions for their suitability before they adopt them.

As a result the style of today's mathematical text is unnecessarily confusing and its notation is unnecessarily clumsy. This is obvious: people tend to commit all the sins they think they can afford to commit. Prior to a commitment to "scaling up" they can hardly be called "sins", but computing's demands on mathematics is sure to cause a "moral" shift between good and bad mathematics. Already now one of the common reactions of the well-trained computing scientist to the average mathematical paper is: "Oh gosh, what a lousy programmer the guy who wrote this must be!" Already now we could start purging the practice of mathematics from the usual little sins of which the computing scientists knows that, at the next level, they will become capital ones. The scaling up of mathematics in general is, of course, a very ambitious topic.

A more modest topic, closer to the field of automatic computing, would be the following. In the case of a soluble problem, what is essentially the same solution can be realized by very different programs. This observation is well-known, but raises all sorts of questions.

A pretty obvious question would be "Which of the possible programs would be the easiest to understand?", but the question is not as innocent as it seems, for what do we mean by "ease of understanding"? Programs are expressed in (what I may here denote by) linguistic structures, and different linguistic structures require different patterns of reasoning for the justification of their usage. Intuitively people tend to prefer the ones they are most familiar with, but we shouldn't confuse "convenient" with "conventional", and should remember that what we call thinking and understanding are not much more than our personal habits, habits that we can replace by others by training ourselves. Some linguistic structures are known to cause troubles in a sense as objective as the statement that decimal arithmetic is "easier" in Arabic than in Roman numerals, but the area as a whole is still a field for investigation and experiment.

Another question in connection with different possible programs, realizing the same solution is the following: "Does a single program suit all our purposes?". The solution might have different relevant aspects, each of which is most manageably reflected in a different program. Who knows? Note that the absence of an example of such a solution does not settle the question: we tend to think about solutions in terms of a program that realizes it, and the mere conception of a solution that requires two or more alternative programs for the adequate reflection of its different aspects might therefore be well beyond our current intellectual abilities! I only raised the last question as an example, in the hope that it may give you some idea of the possible scope of our concerns.

While programming we don't only consider different programs embodying the same solution to a given problem, usually we have to invent the solution as well. Human inventiveness being what it is, we usually invent more than one solution, with the result that we have to choose. When different solutions to the same problem can be shown to be correct by proofs that share part of the argument, is it then possible to realize those solutions by programs that share parts of the text? Those parts of the text could then already be written without completely committing oneself to the solution to be finally adopted; the larger the program, the more important the possibility to postpone such commitments. Equally important is the possibility of postponing the precise choice of the problem to be solved: you really cannot expect your customer --regardless of whether you are your own customer or your customer is someone else-- to have fully made up his mind and stick to it.

In the previous paragraph I have sketched in a very informal fashion some of the flexibility requirements. What the manager sees as "keeping options open" is seen by the scientist as "sharing": different programs sharing code, different proofs sharing arguments, different theories sharing subtheories, and different problems sharing aspects. The need for such "sharing" is characteristic for the design of anything big. The control of such "sharing" is at the heart of the problem of "scaling up" and it is the challenge to the computing scientist or mathematician to invent the abstractions that will enable us to exert this control with sufficient precision.

After having seen how much the catch phrase "the scaling up of mathematics" captures of the challenge posed to us by the existence of modern computers, we are quite naturally led to the question to what extent those very machines --the fast symbol manipulators they are-- could assist us in meeting that challenge. Like all the questions raised in the second part of this talk, also this question will remain unanswered here. I will confine myself to pointing out that the question has two sides.

Firstly we have questions such as "What mechanizable assistance can we think of besides the well-known program transformation and theorem proving systems, and how useful can we expect them to be?", but also, once such a mechanization has been designed "How depends the economics of its usage on the size of the application?". Because I did not define "size", the latter question is vague, but remember that we were interested in "scaling up"! Hence, costs growing quadratically (or worse) as a function of something we would like to increase by several orders of magnitude, would disqualify the tool for the very purpose we had in mind.

Secondly, the answer to what extent computers can assist us in meeting the challenge of "scaling up mathematics" also depends on our interpretation of that challenge. For some mathematicians doing mathematics is a mental activity whose primary purpose is understanding. Such a mathematician is inclined to reject an alleged theorem for which the only available alleged proof is millions of steps long, i.e. orders of magnitude too long to be read and, as he calls it, "understood".

The question "What is Mathematics?" is as unavoidable and as unanswerable as the question "What is Life?". In actual fact I think it's almost the same question.

I thank you for your attention.

Plataanstraat 5
5671 AL NUENEN
The Netherlands

18 November 1979
prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow