ON THE INTERPLAY BETWEEN MATHEMATICS AND PROGRAMMING

E.W. Dijkstra

This talk is delivered under confusing circumstances. And the only way I can think of, of assisting you in not getting confused by these complicating circumstances is describing them explicitly in my introduction. The complication is that, while I would prefer to give a completely technical talk, its moral is heavily loaded from a political point of view: it is a technical talk to be delivered against almost overwhelming political odds. In order to make you understand all this we have to go back to about ten years ago, when Programming Methodology became a topic of explicit concern.

In the history of programming October 1968 has been a turning point. In that month a conference on the topic denoted by the newly coined term "software engineering" was held in Garmisch-Partenkirchen. The conference was sponsored by the NATO Science Committee. The conference was remarkable because a large fraction of the participants had positions so high in their local hierarchy that they could afford to be honest. As a result this was the first sizeable conference at which the existence of the so-called "software crisis" was openly admitted. The gloomy atmosphere of doom at that conference has been a severe shock for some of the participants; some left the place very depressed indeed. The majority, however, left the conference with a feeling of relief, some even in a state of great excitement: it had been admitted *at last* that we did not know to program well enough. I myself and quite a few others had been waiting eagerly for this moment, because now at last something could be done about it. For years we had already been worried by the consequences of the proliferation of error-loaded software, but there is very little point in trying to urge the world to mend its ways as long as that world is still convinced that its ways are perfectly adequate. It was at that conference in Garmisch-Partenkirchen, that the whole climate changed. Now, nearly a decade later, we can only conclude that the excitement was fully justified: it was indeed a turning point in the histo-

ry of programming. Since that conference, programming has never been the same again.

In reaction to the recognition that we did now know how to program well enough, people began to ask themselves how a really competent programmer would look like. What would we have to teach if we wanted to educate a next generation of really competent programmers? This became the central question of the study that later would become known as "programming methodology". A careful analysis of the programmer's task was made, and programming emerged as a task with a strong mathematical flavour. As I have once put it "Programming is one of the hardest branches of applied mathematics because it is also one of the hardest branches of engineering, and vice versa". Why the programming task has such a strong mathematical flavour is something I shall indicate later.

A lower bound for what the adequate education of a really competent programmer should comprise was very convincingly established, but it was *not* an easy message to sell, because it demonstrated by necessity the total inadequacy of the education of what is known as "the average programmer". The world today has about a million "average programmers", and it is frightening to be forced to conclude that most of them are the victims of an earlier underestimation of the intrinsic difficulty of the programmer's task and now find themselves lured into a profession beyond their intellectual capabilities. It is a horrible conclusion to draw, but I am afraid that it is unavoidable.

The conclusion that competent programming required a fair amount of mathematical skills has been drawn on purely technical grounds and, as far as I know, has never been refuted. On emotional grounds which are only too understandable, many people have refused to draw the conclusion, and the conclusion is opposed to, not because its validity is challenged, but because its social consequences are so unpleasant.

The situation is immensely aggravated by changes in attitude towards science and technology in general, that took place during the sixties. In that decade we have seen a growing distrust of technology, a disillusion with science, which by the end of that decade caused political outbursts from which most universities haven't fully recovered yet.

For those who had hoped that the explosive growth of universities and other research establishments would automatically bear fruits in proportion to that growth, the results have indeed been disappointing, because, while

the quantity grew, the average quality declined. Browsing through a scientific journal or attending a conference is nowadays rather depressing; there is no denying it: there is just an awfull lot of narrow mediocrity, of downright junk even. Many people seem to have failed to see, that it was not science itself, but only the explosive growth of the institutions that was to blame. Throwing away the child with the bathwater, they have declared war on science in its best tradition. They are openly antiscientific, antiacademic, very much against rigour and formal techniques, and they propose to be agressively creative, gloriously intuitive and nobly interdisciplinary instead. The cruel love of perfection and excellence, that used to characterize the hard sciences, are but elitist relics to be abolished as quickly as possible, and progressive from now onwards shall mean soft. The political slogans of the late sixties cast these views in a jargon that is still alive and still causes confusion.

The result of all this is that the message that "software", in spite of its name, requires a very hard discipline, is in many environments now politically unacceptable, and therefore fought by political means. In characteristically anonymous blurbs in periodicals of the Computer Weekly variety I find myself under political attack. "Dijkstra articulates the voice of reaction" is a mild one. "I am inclined to view Dijkstra [...] as intellectual product of the Germanic system" is much worse. And I arouse the "suspicion that [my] concepts are the product of an authoritarian upbringing" coming as I do from a country having "social philosophies touched by authoritarianism and the welfare state" etc. Nice is also the choice of adjectives when my efforts are described as "directed into turning a noble art into a rigid discipline". The first time I found myself confronted with the opinion that adhering to a formal discipline hampers creativity I was completely baffled, because it is absolutely contrary to my experience and the experience of the people I have worked with. I found the suggestion so ludicrous that I could not place it at all: it is so terribly wrong. Since then I have learned that as symptom of a political attitude it is quite well interpretable.

Having thus -I hope- cleared the sky from political encumbrances, I shall now turn to the technical part of my talk.

Why is programming intrinsically an activity with a strong mathematical flavour? Well, mathematical assertions have three important characteristics.

1) Mathematical assertions are always general in the sense that they are applicable to many -often even infinetely many- cases: we prove something for *all* natural numbers or *all* nondegenerate Euclidean triangles.
2) Besides general, mathematical assertions are very precise. This is already an unusual combination, as in most other verbal activities generality is usually achieved by vagueness.
3) A tradition of more than twenty centuries has taught us to present these general and precise assertions with a convincing power that has no equal in any other intellectual discipline. This tradition is called Mathematics.

The typical program computes a function that is defined for an incredibly large number of different values of its arguement; the assertion that such and such a program corresponds to such and such a function has therefore the generality referred to above.

Secondly: the specification of what a program can achieve for us must be pretty precise, if it is going to be a safe tool to use. Regarded as a tool its usage can only be justified by an appeal to its stated properties, and if those are not stated properly its usage cannot be justified properly. And here we have the second characteristic.

Thirdly: the assertion that such and such a program corresponds to such and such a function, although general and precise, is not much good if it is wrong. If the program is to be regarded as a reliable tool, our least obligation is a convincing case, that that assertion is correct. That program testing does *not* provide such a convincing case is well-known. The theoretically inclined can deduce this from the indeed incredibly large number of different argument values for which the function is typically defined; the more experimentally inclined can conclude this from more than twenty years of experience in which program testing as main technique for quality control has not been able to prevent the proliferation of error-loaded software. The only alternative that I see is the only alternative mankind has been able to come up with for dealing with such problems, and that is a nice convincing argument. And that is what we have always called Mathematics.

Here we touch upon the major shift in the programmer's task that took place during the last ten years. It is no longer sufficient to make a program of which you hope that it is correct -i.e. satisfies its specifications- you must make the program in such a way that you can give a convincing argument for its correctness. Superficially it may seem that this shift

has made the task of the poor programmer only more difficult: besides making a program he has to supply a correctness argument as well. It may indeed be hard to supply a nice correctness argument for a *given* program; if, however, one does not add one's correctness concerns as an afterthought, but thinks about the correctness argument right at the start, the correctness concerns have proved to be of great heuristic value. And the wise programmer now develops program and correctness argument hand in hand; as a matter of fact, the development of the correctness argument usually runs slightly ahead of the development of the program: he first decides how he is going to prove the correctness and then designs the program so as to fit the next step of the proof. That's fine.

You may think that I have introduced a more serious difficulty by stating that the programmer should make his program in such a way that he can give "a convincing argument" for its correctness. Convincing to whom? Well, of course, only to those who care. But couldn't those have very, very different notions of what to regard as "convincing"? Has the programmer to provide as many different arguments as there may be people caring about the correctness of his program? That would make his task clearly impossible.

The task is, indeed, impossible as long as we don't distinguish between "conventional" and "convenient". What different people from different parts of the world have been used to varies so wildly, that it is impossible to extract a guiding principle from trying to present your argument in the most "conventional" way: their usual patterns of thinking are most likely inadequate anyhow. About convenience of a notation, about effectiveness of an argument, about elegance of a mathematical proof, however, I observed among mathematicians a very strong consensus -the consensus was, as a matter of fact, much greater than most of the mathematicians I spoke suspected themselves- and it is this consensus among mathematicians that has proved to be a very valuable guiding principle in deciding towards what type of "convincing argument" the programmer should be heading.

Let me now try to sketch to you the type of mathematics involved in arguing about programs. One way of viewing a program is as the rules of behaviour which can be followed by an automatic computer, which is then said "to execute" the program. The process taking place when a computer executes a program is called a "computation", and a computation can be viewed as a time-sequence or a long succession of different machine states. The part of the machine in which its current state is recorded is called the store -or:

the memory-; the store is very large because it must be able to distinguish between a huge number of different states.

In arguing about programs we have to characterize the set of machine states that are possible at various stages of the computational process. Individual states are characterized by the values of variables in very much the same way as the position of a point in a plane can be characterized by the value of its coordinates in a well-chosen coordinate system. There are in this analogy only two differences: while the coordinates in the Euclidean plane are usually viewed as continuous, the variables characterizing the state of the machine are discrete variables that can only take on a finite number of different values. And secondly: while in Euclidean plane geometry two coordinates suffice to fix the position of a point, in computations we typically need thousands or millions of different variables te record the current machine state.

In spite of the fact that that last difference is a drastic one, the analogy is yet a useful one. Everybody familiar with analytic geometry knows how specific figures, lines, circles, ellipses etc. can be characterized by *equations*: the figures are regarded as the subset of the points whose coordinates satisfy the equation. The analogy to the figure in analytic geometry is the subset of possible states at a certain point of progress of the computation, and in analogy to analytic geometry, such a subset is characterized by an equation: the subset comprises all states of the machine in which the values of the variables satisfy that equation.

The analogy can even be carried a little bit further: we all know how the ease with which a proof in analytical geometry can be carried out often depends on the choice of our coordinate system. The program designer has a similar freedom when he chooses the conventions according to which the variables he introduced shall represent the information to be manipulated. He can use this freedom to speed up the computation; he can also use it to simplify the equations characterizing the sets of states he is interested in. If he is lucky, or gifted, or both, his choice of representation serves both goals.

So much for the analogy; now for the difference. The number of variables he is dealing with is much larger than the two coordinates of plane geometry, and the subsets of machine states he needs to characterize have very seldomly an obvious regularity as the straight line, the circle, and the ellipse that analytic geometry is so good at dealing with. This has

two immediate consequences.

First of all we need a much richer framework and vocabulary in which we can express the equations than the simple algebraic relations that carry analytic geometry. The framework is provided by the first-order predicate calculus, and the vocabulary by the predicates the programmer thinks it wise to introduce. That the first-order predicate calculus was the most suitable candidate for the characterization of sets of machine states was assumed right at the start; early experiences, however, were not too encouraging, because it only seemed practicable in the simplest cases, and we discovered the second consequence: the large number of variables combined with the likely irregularity of the subsets to be characterized very quickly made most of the formal expressions to be manipulated unmanageably long.

Let me put it in other words. The programmer is invited to apply the first-order predicate calculus; I am even willing to make a stronger statement: not knowing of any other tool that would enable to do the job, the programmer *must* apply the first-order predicate calculus. But he has to do so in an environment in which he is certain to create an unmanageable mess unless he *carefully* tries to avoid doing so (and even then success is not guaranteed!). He has to be constantly consciously careful to keep his notation as adequate and his argument as elegant as possible. And it is only in the last years that we are beginning to discover what that care implies. Let me give you a simple example to give you some feeling for it.

To begin with we consider a finite undirected graph at each vertex of which a philosopher is located: philosophers located at vertices that are directly connected by one edge of the graph are called each other's neighbours and no philosopher is his own neighbour. For the time being the life of a philosopher exists of an endless alternation of two mutually exclusive states, called "thinking" and "tabled".

In our next stage we allow edges to be marked or not, a marked edge meaning that the two philosophers at its two ends are both tabled, more precisely

P1: For any pair (A, B) of neighbours
"both A and B are tabled" = "the edge between A and B is marked".

We assume that the system is started in an initial state in which

1) all edges are unmarked

2) all philosophers are thinking.

As a result, P1 initially holds. Therefore P1 will continue to hold indefinitely, provided no philosopher transition from thinking to tabled introduces a violation of it. This is obviously achieved by associating with these transitions the following "point actions" -where no two different point actions are assumed to take place simultaneously-

T1: < mark the edges connecting you to tabled neighbours and switch from thinking to tabled >

T2: < unmark your marked edges and switch from tabled to thinking >.

The first transition now introduces a mark for every pair of tabled neighbours introduced by it, the second one removes a mark for every pair of tabled neighbours disappearing as a result of it. With these conventions the permanent truth of P1 is guaranteed.

From the above we see that a mark on the edge between the neighbours A and B has either been placed by A or by B. In our next stage we shall indicate which of the two has placed the mark by representing a marked edge between A and B by a directed edge, i.e. by placing an arrow along the edge. In this representation relation P1 is rephrased as

P1: For any pair (A, B) of neighbours
"both A and B are tabled" = "the edge between A and B is directed".

The direction of the arrow is fixed, by rephrasing the transitions as

T1: < direct arrows pointing towards your tabled neighbours and switch from thinking to tabled >

T2: < make all your edges undirected and switch from tabled to thinking >.

We observe that transitions T1 create arrows and only transitions T2 destroy them. More precisely: each arrow is created as an outgoing arrow of its creator, hence,

a philosopher without outgoing arrows remains without outgoing arrows until it performs itself its *own* transition T1.

We now subdivide the state "tabled" into the succession of two substates "hungry" followed by "eating", where the transition is marked by the observation of absence of outgoing arrows, more precisely

"philosopher A is tabled" = "philosopher A is hungry or eating"

and the life of a philosopher now consists of a *cyclic* pattern of transitions

T1: < direct arrows pointing towards your tabled neighbours and switch from thinking to hungry >

T1.5: < observe that you have no outgoing arrows and switch from hungry to eating >

T2: < remove all your incoming arrows and switch from eating to thinking >

and we establish the permanent truth of

P2: For any philosopher A we have
"philosopher A has no outgoing arrows" or "philosopher A is hungry".

In transition T1 the first term P2 may become false, but the second one becomes certainly true; in transition T1.5 the second term becomes false at a moment when the first term is true, a truth that cannot be destroyed by the other philosophers. In T2 the fact that initially the philosopher is eating tells us in combination with P2 that its arrows, if any, must be incoming arrows; hence, removal of your incoming arrows is the same as removal of all your arrows.

Relations P1 and P2 guarantee that no two neighbours can be eating simultaneously: if they were, they would both be tabled, hence there would be an arrow between them (on account of P1), for one of them it would be an outgoing arrow, but P2 excludes that an eating philosopher, which by definition is not hungry, has outgoing arrows.

(In addition we can prove that if the graph is finite and each eating period for each philosopher is finite, then each hungry period for each philosopher will be finite. This follows from the fact that the arrows never form a directed cyclic path.)

The way in which the above argument has been described illustrates one of the aspects of the "care" which is becoming typical for the competent programmer: "step-wise refinement" is one of the catchwords. Note that we have started the argument in terms of the still very simple concepts "tabled" and "marked". Only after the exhaustion of these two concepts, the state "marked" was split up into two mutually exclusive substates as represented by the two possible directions of an arrow along the edge. And only when the consequences of that refinement had been explored, the state "tabled" was subdivided into two mutually exclusive states, viz. "hungry" and "eating".

In the simple example shown such a cautious approach may seem exaggerated, but for the trained programmer it becomes a habit. In a typical program so many different variables are manipulated that the programmer would

lose his way in his argument if he tried to deal with them all at once. He has to deal with so many concerns that he would lose his way if he did not separate them fairly effectively. He tries to keep his arguments simple compared to the final program by abstracting from all sorts of details that can be filled in later.

In yet another respect the above argument is typical. I did not tell you the original problem statement, but that was phrased as a synchronization problem, in which no two neighbours were allowed to eat simultaneously. The notion "hungry" has to be invented by the programmer; and then the argument is introduced by abstracting from the difference between "hungry" and "eating", in terms of the notion "tabled" that did not occur in the original problem statement at all. Such abstractions *must* be performed: instead of "tabled" one can say "hungry" or "eating", but the translation of "a pair of tabled neighbours" gives you some hint of the clumsiness thus engendered.

One last detail worth noticing is provided by our arrows. We had to introduce two different forms of marking: we could have done that with colours, say red edges and blue edges, but then we would have lost that my incoming arrows are my neighbours outgoing arrows, and the whole argument would have lost its clarity.

So much for the care needed to keep the arguments manageable: we can summarize it by stating that in programming mathematical elegance is not a dispensable luxury, but a matter of life and death.

In the example sketched the argument could be rendered nicely and compactly essentially thanks to the introduction of the proper nomenclature, but quite often more drastic steps have to be taken. In order to formulate the equations characterizing sets of possible machine states it is quite often necessary to change the program by the insertion of additional operations on so-called "auxiliary variables". They are not necessary for the computation itself, they are hypothetical variables whose values we can view as being changed in the course of the computational process studied. They record some aspect of the progress of the computation that is not needed for the answer, but for the argument justifying the program. Their values can appear in the characterizing equations in terms of which the correctness argument is couched. The introduction of the appropriate auxiliary variables is a next step in the progress of "choosing an adequate nomenclature"; the role of the auxiliary variables in proofs of program

correctness is very similar to the role of auxiliary lines or points in geometrical proofs, and their invention requires each time a similar form of creativity. This is one of the reasons why I as a computing scientist can only regret that the attention paid to Euclidean geometry in our secondary school curricula has been so drastically reduced during the last decades.

In a recent correctness proof I had to go still one step further. I had to introduce auxiliary variables, but their values did not occur directly in our characterizing equations: in those equations occurred terms which had to be defined as the minimal solution of two sets of equations in which the auxiliary variables appeared as constants. As far as I am aware, that proof was the first one of its kind, but its discovery was a pure joy. It showed a counterexample to the commonly held but erroneous belief that formal correctness proofs for programs are only belabouring the obvious; it showed how the first-order predicate calculus was an indispensable and adequate tool, but, most important of all, it showed how a careful analysis of the syntactic structure of the predicates quite naturally led to all the additional logical gear to be invented.

In the interplay between mathematics and programming during the last ten years programming as an intellectual discipline has clearly been at the receiving end. A new area of intellectual activity has been discovered to be amenable to mathematical treatment, and thanks to the introduction of mathematical techniques we can now design programs that are an order of magnitude better than the ones we could design ten years ago. In the past the discovery of a new area of applicability of mathematics has always influenced and stimulated mathematics itself, and it is reasonable to wonder about the question what influence on mathematics may be expected this time.

I expect that the influence will be very wholesome. The programmer applies mathematical techniques in an environment with an unprecendented potential for complication; this circumstance makes him methodologically very, very conscious of the steps he takes, the notations he introduces etc. Much more than the average mathematician he is explicitely concerned with the effectiveness of this argument, much more than the average mathematician he is consciously concerned with the mathematical elegance of his argument. He simply has to, if he refuses to be drowned in unmastered complexity. From the programmer's exposure and experience I can expect only one influence on mathematics as a whole: a great improvement of the taste with which formal methods are applied. This improvement may very well turn

out to be drastic. In texts about the philosophy of science from the first half of this century it is quite common to encounter a postulated antagonism between formal rigour on the one hand and "understandability" on the other. Already now, whenever I see such a text it strikes me as hopelessly out of date, arguing as it does against formal rigour instead of against ugliness: in those days the two were evidently often regarded as synonymous. And I have some indication that this improvement in taste is not only the dream of an optimist. I have conducted a little experiment with students from all over the world, in which I asked them to prove a nice little theorem from number theory that, although everyone can understand what the theorem states, happens to be unknown: the mathematicians with programming experience did markedly better than the mathematicians without that experience.