Copyright Notice

The manuscript was published as pages 324–328 of

Edsger W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982. ISBN 0–387–90652–5.

Why naive program transformation systems are unlikely to work.

Look how carefully the title has been worded! No developer of a program transformation system needs to feel offended, for I have given him two escapes. First, I am not arguing an impossibility, but only an unlikeliness --and we know that all startling advances have been made against seemingly overwhelming odds, don't we?-- ; second, he has the option to declare that the program transformation system he is developing is not "naive" in the sense that I shall make more precise below.

* * *

I take the position that a serious programmer has at least two major concerns, viz. that his programs are correct and are efficient. And from existing software we can deduce that neither of these two concerns is a trivial one.

For years I have argued what I still believe, namely that, when faced with different concerns, we should try to separate them as completely as possible and deal with them in turn. In the case of the concerns for correctness and for efficiency this separation has been achieved up to a point. It is possible to treat the problem of program correctness in isolation from the problem of efficiency in the sense that we can deal with the correctness problem temporarily even ignoring that our program text also admits the interpretation of executable code. It is also possible to investigate the various cost aspects of program execution independent of the question whether such execution of the program will produce a correct result.

Presented as in the previous paragraph the separation sought seems to have been found. It is true that the separation is reachable as far as the program text itself is concerned; in the process of composing the text, however, the separation is less marked. There does exist a formal discipline that, when adhered to, cannot lead to an incorrect program. In its application, however, we have a great amount of freedom, and in the choice how to apply the discipline ensuring correctness, the designer always makes up his mind by considering his other concerns, such as efficiency. In other words, the more rigorous the concerns have been separated with respect to the program text itself, the more

schizophrenic the act of program compositon becomes: the programmer still remains a jack of many trades, switching all the time --and at a high frequency!-- between various roles, whose differences have only become more and more marked over the last decade.

* * *

I have been introduced to program transformation systems as to overcome the need for such a schizophrenic programmer behaviour. A number of so-called "semantics preserving program transformations" have been discovered. Each such transformation, when applicable and applied to a program A generates a new program A' that, when executed, will produce the same result as the original program A , the difference being that the costs of execution of A and of A' may differ greatly. Program A' may also be derived by successive applications of a sequence of such transformations.

It was the discovery of (sequences of) such transformations that supported the idea of --what I call: naive-- program transformation systems. When using such a system for the development of a program, this development was envisaged to take place in two successive, clearly and rigorously separated stages.

In the first stage the programmer would only be concerned with program correctness: unencumbered by efficiency considerations he would write a program, the correctness of which could be established as easily as possible. In the ideal case, the program's correctness would be trivial to establish.

In the second stage --which in the dreams of some could or should be conducted by a different person, unfamiliar with the original problem-- the correct but inefficient program would be subjected to semantics preserving transformations from a library, until the program had become efficient as well. (At the moment this dream was dreamt, the available library of acknowledged transformation was admittedly still somewhat small, but it was constantly growing and hopes were high.)

* * *

When such systems were proposed to me I was very sceptical, but I was

mainly so for a purely personal reason and accidental circumstance. Their advocates tried to convince me of the viability of their approach by composing according to their proposed method a program I had published myself. In their demonstrations, stage two required about ten pages of formal labour, while stage one had taken them between one day and one week.

It so happened that their demonstrations were not very convincing for me, because, heading schizophrenicly towards a correct and efficient solution, I myself had solved the whole problem (without pencil and paper) in fifteen minutes. (It was the evident effectiveness of the heuristics applied that had prompted that publication: the problem itself was one of the kind I could not care less about.)

At the time I was not worried so much about the ten pages of stage two, as it was clear that most of it could be mechanized and never need to see the light of day. I was much more worried about the discrepancy between one or several days for stage one on the one hand, and fifteen minutes for the whole job on the other, and I remember voicing this latter worry at a meeting of the IFIP Working Group 2.3 on "Programming Methodology".

One of the members --a pioneer in program transformations-- suggested a possible explanation for the observed discrepancy: as programmers we had in the past been so terrorized by efficiency concerns that it was very difficult for us to come up with a trivially correct solution, no matter how grossly inefficient. He supported his explanation by stating a problem and presenting a solution for it that, indeed, was so ridiculously inefficient that it would never have entered my mind.

I was struck by his argument --otherwise I wouldn't have remembered it!-- , he made me doubt but could not convince me. The possible explanation for the discrepancy that I had considered was that, by ignoring efficiency considerations, the "admissible solution space" had become cumbersomely large: I felt that the efficiency considerations could provide a vital guiding principle. It seemed a draw and for the next eight months I did not make up my mind any further about the chances of success for naive program transformation systems.

* * *

All the above was introduction. After the closing ceremony of IFIP77 in Toronto I had dinner with Jan Poirters and Martin Rem, and in a conversation about the role of mathematics in programming I ventured the conjecture that often an efficient program could be viewed as the successful exploitation of a mathematical theorem. I presented an efficient program as a piece of logical brinkmanship in which a cunning argument could show that the computational labour performed would be just enough for reaching the answer.

I came up with the example of the shortest subspanning tree between N points. There exists a simple one-to-one correspondence between the N^{N-2} different subspanning trees between N points and the N^{N-2} different numbers of N-2 digits in base n. A naive computation A could therefore generate all N^{N-2} trees and select the shortest one encountered. But we know that there exists an efficient algorithm A' whose computation time is proportional to N^2. But the only way in which I can justify the latter algorithm is by using (a generalization of) the theorem that of the branches of the complete graph that meet in a single point, the shortest one is also a branch of the shortest subspanning tree.

In confirmation of our experience that everything of significance in computing science can be illustrated with Euclid's algorithm, Martin Rem came with that example. In order to compute the greatest common divisor of a positive X and Y, the correct algorithm A constructs a table of divisors of X, then a table of divisors of y, then the intersection of the two tables, and from that (finite and nonempty) intersection the greatest value is selected. But good old Euclid knew already algorithm A' which I can only justify by appealing to (a generalization of) the theorem that gcd(x, y) = gcd(x, y-x).

The next week David Gries told me about a speeding up of the Sieve of Eratosthenes --another classic!-- for generating a table of prime numbers, a job for which many inefficient but correct algorithms can be created, e.g.

```
y, p := 1, 1;
do p < N → p:= p + 1; do gcd(p,y) ≠ 1 → p:= p + 1 od;
          print(p); y:= y * p
od
```

David's program, however, relied on the theorem that there exists a prime number between n and $2n$.

In the meantime I have thought of a fourth example. The branches of a subspanning tree between N points provide a unique path between any of the two points and we can define the sum of the branches of such a path to be the "distance" between those two points. Which is the point pair with the maximum distance from each other? The simple algorithm A determines all $N(N-1)/2$ distances and selects the longest encountered. The efficient algorithm A' uses the theorem that for an arbitrary point y the point x with the maximum distance from y is one of the end points of the longest path. We then determine the point z with the maximum distance from x, and the pair (x, z) is our answer.

The question is now what our chances are of deriving the efficient program A' by applying (mechanizable) transformations from a finite library to the original program A? Because the transformations are semantics preserving, program A' is correct if program A is. The correctness proof for A --a proof which, ideally, is almost trivial--, together with the derivation path from A to A', constitutes a correctness proof for A'. In none of the examples given the theorem with which we proved the correctness of A' seemed unnecessarily strong, i.e. from the given correctness of A' the corresponding theorem seems in each case simply derivable. The supposed derivation path from A to A' therefore contains not only the major part of the justification of A', but also of the proof of the mathematical theorem that we used to justify program A' directly.

All our experience from mechanized mathematics tells us that therefore the derivation paths from A to A' --if, with a given library, they exist at all-- can be expected to be long and to be difficult to find. Extending the library is only an improvement as long as the library is still very small: using a large library will be exactly as difficult as commanding a large body of mathematical knowledge. Furthermore, each intermediate product on the derivation path from A to A' must be a program that is semantically equivalent to A; this seems a constraint for which I can find no analogue in normal mathematical reasoning, and for many triples $< A, A',$ library $>$ it may make even the existence of such a derivation path questionable?

The stated hope that, once our system of mechanized program transformations is there, stage two can be left to a sort of "technical assistant" that need not know anything about the original problem and its underlying mathematics, but only needs to know how to operate the transformation system, now seems to me unwarranted. And if that hope is expressed as a claim, that claim now seems to me just as misleading as most advertizing.

I do not exclude the possibility that useful program transformation systems of some sort will be developed --it may even be possible to derive some of the efficient algorithms I mentioned above-- , but I don't expect them to be naive: the original goal of allocating the mathematical concern about correctness and the engineering concern about execution costs to two distinct, well-separated stages in the development process seems unattainable. It was good old Euclid who warned king Ptolemy I:

"There is no 'royal road' to geometry." ;

and those who think that that warning does not apply to <u>them</u>, will be reminded of it the hard way.....

<u>Acknowledgement</u>. The argument displayed above contains enough loose expressions --such as "a major part of the proof"-- to be regarded as fishy. I am not even myself perfectly sure of its convincing power. (How is <u>that</u> for a loose expression?) I therefore gratefully acknowledge the opportunity provided in Niagara-on-the-Lake, Aug.1977, to confront members of IFIP WG2.3 with it and to solicit their comments. Although I found my feelings confirmed, it goes without saying that none of them can be held responsible for the views expressed in the above. I also thank Jan Poirters and Martin Rem for their contribution to a pleasant, yeah even memorable dinner. (End of acknowledgement.)

Plataanstraat 5
5671 AL NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow