Copyright Notice

FORMAL TECHNIQUES AND SIZEABLE PROGRAMS

Edsger W.Dijkstra
Burroughs
Plataanstraat 5
NL-4565 NUENEN
The Netherlands

By now we know quite convincing, quite practical and quite effective methods of proving the correctness of a great number of small programs. In a number of cases our ability is not restricted to a posteriori proofs of program correctness but even encompasses techniques for deriving programs that, by virtue of the way in which they have been derived, must satisfy the proof's requirements.

This development has taken place in a limited number of years, and has changed for those who are familiar with such techniques their outlook on what programming is all about so drastically, that I consider this development both fascinating and exciting: fascinating because it has given us such a new appreciation of what we already knew how to do, exciting because it is full of unfathomed promises.

This development is the result of a very great number of experiments: experiments in programming, in axiomatizing, and in proving. It could never have taken place if the researchers in this field had not shown the practical wisdom of carrying out their experiments with small programs. As honest scientists they have reported about their actual experiences. This, alas, has created the impression that such formal techniques are only applicable in the case of such small programs.

Some readers have exaggerated and have concluded that these techniques are primarily or exclusively applicable to so-called "toy problems". But that is too great a simplification. I do not object to describing Euclid's Algorithm for the greatest common divisor as a "toy problem" (in which capacity it has been a very fertile one!). But I have also seen perfectly readable and adequate formal treatments of much less "toyish" programs, such as a binary search algorithm and a far from trivial algorithm for the computation of an approximation of the square root, which would be ideal for a microprogram in a binary machine. I call this last algorithm "far from trivial" because, although it can be described in a few lines of code, from the raw code it is by no means obvious what the algorithm accomplishes.

The question that I would like to address here is what we may expect beyond those "small examples". Hence the adjective "sizeable" in my title.

The crude manager's answer to my question is quite simple: "Nothing.". He will argue that difficult problems require large programs, that large programs can only be written by large teams which, by necessity, are composed of people with, on the average, n-th rate intellects with n sufficiently large to make formal techniques totally unrealistic.

My problem, however, is that I don't accept that answer, as it is based on two tacit assumptions. The one tacit assumption is that difficult problems require large programs, the second tacit assumption is that with such a Chinese Army of n-th rate intellects he can solve the difficult problem. Both assumptions should be challenged.

On challenging the second assumption I don't need to waste many words: the Chinese Army approach --also called "the human wave"-- has been tried, even at terrific expense, and the results were always disastrous. OS/360 is, of course, the best known example of such a disaster, but please, don't conclude from NASA's successful moonshots that in other cases it has worked. There is plenty of evidence that the data processing associated with these NASA ventures was full of bugs, but that the total organization around it was, however, so redundant that the bugs usually did not matter too much. In short, there is plenty of experimental evidence that the Chinese Army approach does not work; and as a corollary we may conclude that the perfection of Chinese Army Generals is a waste of effort. At the end of my talk I hope that you will agree with me that, in order to reach that conclusion, said experimental evidence was superfluous, because a more careful analysis of the tasks at hand can teach us the same.

* * *

For my own instruction and in order to collect material for this talk I conducted and experiment that I shall describe to you in some detail. I do so with great hesitation because I know that, by doing so, I shall sow the seed of misunderstanding. The problem of a speaker is that, if he does not give examples, his audience does not know what he is talking about, and that, if he gives an example, his audience may mistake it for his subject! In a moment I shall describe to you my experiment and you will notice that it has to do with syntactic analysis, but, please, remember that syntactic analysis is not the subject of my talk, but only the carrier of my experiment for which I needed an area for computer application in

which I am most definitely not an expert.

I wrote a paper with the title "A more formal treatment of a less simple example.". Admittedly it was still not a very large example: the final solution consisted of four procedures, of which, in beautiful layout with assertions inserted, three were only 7 lines long and the last one 18 lines. But the whole document is 19 typed pages, i.e. about 14 times as long as the raw code. It took me several weeks of hard work to write it, and when it was completed I was grateful for not having been more ambitious as far as size was concerned. It dealt with the design of a recognizer for strings of the syntactic category < sent > , originally given by the following syntax:

```
< sent > ::= < exp > ;                                                    (1)
< exp >  ::= < term > | < exp > + < term > | < exp > - < term >
< term > ::= < prim > | < term > * < prim >
< prim > ::= < iden > | ( < exp > )
< iden > ::= < letter > | < iden >< letter >
```

That was all!

My first experience was that, in order to give a more precise statement about the string of characters that would be read in the case that the input was not an instance of < sent > , I needed new syntactic categories, derived from (1) and denoting "begin of...": for each syntactic category < pqr > I needed the syntactic category < bopqr > , characterizing all strings that either are a < pqr > or can be extended at the right-hand side so as to become a < pqr > or both.

```
< bosent > ::= < sent > | < boexp >                                       (2)
< boexp >  ::= < boterm > | < exp > + < boterm > | < exp > - < boterm >
etc.
```

(In an earlier effort I had also used the notion "proper begin of a < pqr > ", i.e. at the right-hand side extensible so as to become a < pqr > but not a < pqr > by itself. This time I obtained a simpler and more uniform treatment by omitting it and only using "begin of..." as derived syntactic categories.)

The next important step was the decision to denote the fact that the string K belongs to the syntactic category < pqr > by the expression:

pqr(K) .

This decision was an immediate invitation to rewrite the syntax as follows:

```
< sent > ::= < exp >< semi >                                        (3)
   < semi > ::= ;
< exp >  ::= < term > | < exp >< adop >< term >
   < adop > ::= + | -
< term > ::= < prim > | < term >< mult >< prim >
   < mult > ::= *
< prim > ::= < iden > | < open >< exp >< close >
< iden > ::= < letter > | < iden >< letter >
   < open > ::= (
   < close > ::= )
```

The invitation, however, was only noticed after I had dealt with the first line of the syntax, dealing with < sent > ; when dealing with < exp > , it was the occurrence of both the + and the - that induced the introduction of < adop > , because without it my formulae became full of insipid duplication. It was only then that I discovered that the boolean procedure "semi(x)" --only true if the character x is a semicolon-- and the other boolean procedures that I needed for the classification of single characters were a specific instance of the convention that introduced "pqr(K)" . Finally I realized that the usual BNF , as used in (2), is an odd mixture in the sense that in the productions the characters stand for themselves; in (3) this convention is restricted to the indented lines.

A next important decision was to denote for strings (named K, L, ...) and characters (named x, y, ...) concatenation simply by juxtaposition, e.g. KL , Ky yLx , etc. Now we could denote the arbitrary nonempty string by yL or Ly and could derive from our syntax formulae like

$$(exp(L) \ \underline{and} \ semi(y)) \Rightarrow sent(Ly) \quad .$$

It also enabled me to define the "begin of...":

$$bopqr(K) = (\underline{E}\ L:\ pqr(KL)) \quad .$$

I mention the apparently trivial and obvious decision to denote concatenation by juxtaposition explicitly, because in the beginning my intention to do a really neat formal job seduced me to introduce an explicit concatenation operator. Its only result was to make my formulae, although more impressive, unnecessarily unwieldy.

From my earlier effort I copied the convention to express post-conditions in terms of the string of characters read. With "S" defined as the string of input characters "read" --or "moved over" or "made invisible"-- by a call of "sentsearch",

and with "x" defined as the currently visible input character, we can now state the desired post-condition for our recognizer "sentsearch":

Rs(S, x, c): bosent(S) and non bosent(Sx) and c = sent(S) (4)

The first term expresses that not too much has been read, the second term expresses that S is long enough, and the last term expresses that in the global boolean "c" --short for "correct"-- the success or failure to read a < sent > from the input should be recorded.

In short, we treat S and x as variables (of types "character string" and "character" respectively) that are initialized by a call of sentsearch . I mention this explicitly, because for a while we departed from that convention, and did as if the "input still to come" were defined prior to the call of sentsearch. We tried to derive from our post-condition weakest pre-conditions in terms of the "future" input characters, and the result was a disaster. At some time during that exercise we were even forced to introduce a deconcatenation operator! The trick to regard as "post-defined output" what used to be regarded as "pre-defined input" cannot be recommended warmly enough: it shortened our formulae with a considerable factor and did away with the need for many dummy identifiers.

Another improvement with respect to our earlier effort was a changed interface with respect to the input string. In my earlier trial I had had as a primitive to read the next character

x:= nextchar

where "nextchar" was a character-valued function with the side-effect of moving the input tape over one place. (If S is the string of characters read, the above assignment to x should be followed implicitly by the "ghost statement" S:= Sx .) Prior to the first x:= nextchar , the value of the variable x was supposed to be undefined. In the new interface, where x is the currently visible character and S the string of characters no longer visible, I chose the primitive "move", semantically equivalent to the concurrent assignment

S, x := Sx, new character .

This minor change of interface turned out to be a considerable improvement! In the new interface, the building up of S lags one character behind compared with the old interface. Formula (4) shows how we can now refer --via concatenation-- to two strings, one of which is a character longer than the other. With the old interface we would have needed a notation for a string one character shorter than S , some-

thing so painful that in my earlier effort a different specification for sentsearch was chosen, with the old interface more easily described, but logically less clean than (4).

I wanted to write a body for sentsearch in terms of a call on expsearch and the boolean primitive semi(x) which was assumed to be available. I wished to do so only on account of the syntax for < sent > and discovered that I only could do so under the assumption --to be verified later when the full syntax was taken into account-- that

$$\text{sent}(L) \Rightarrow \underline{\text{non}}\ (\underline{\text{E}}\ y:\ \text{bosent}(Ly)) \qquad (5)$$

would hold. Confronting this with the specification (4) we conclude that if sentsearch establishes a final state with c = true , i.e. sent(S) , the second term --non bosent(Sx)-- is true for all values of x : in other words, postulate (5) states that the end of an instance of the syntactic category < sent > can be established "without looking beyond".

We assume the availability of a primitive expsearch . Defining "E" to be the string of input characters mover over by it, it establishes, analogous to (4):

$$\text{Re}(E, x, c):\quad \text{boexp}(E)\ \underline{\text{and non}}\ \text{boexp}(Ex)\ \underline{\text{and}}\ c = \text{exp}(E) \qquad (6)$$

Called by sentsearch, it implies S:= SE (as "move" implies S:= Sx). A possible body for sentsearch is now:

```
proc sentsearch: {S = empty string}
                 expsearch {Re(S, x, c)};
                 if non c → {Rs(S, x, c)} skip {Rs(S, x, c)}
                  [] non semi(x) → {Rs(S, x, false)} c:= false {Rs(S, x, c)}
                  [] c and semi(x) → {Rs(Sx, y, c)} move {Rs(S, x, c)}
                 fi {Rs(S, x, c)}
corp
```

For its correctness proof I needed three theorems:

Theorem 1. (Re(L, x, c) and non c) => Rs(L, x, c)

Theorem 2. (Re(L, x, c) and non semi(x)) => Rs(L, x, false)

Theorem 3. (Re(L, x, c) and c and semi(x)) => Rs(Lx, y, c)

The proofs of these three theorems and also of

$$\text{boexp}(L) \Rightarrow \underline{\text{non}}\ \text{sent}(L)$$

that I needed in these proofs took more than one-and-a-half page.

In the meantime the first 6 of the 19 pages had been written. The primitive

expsearch asked for another three theorems to be proved and was finished 4 pages later; by analogy termsearch took only half a page; the primitive primsearch required another six theorems to be proved and was completed 6 pages later. The remaining two-and-a-half page were needed to prove assumption (5) and the similar

$$(\text{term}(L) \text{ and } \text{adop}(y)) \Rightarrow \text{non } \text{boterm}(Ly)$$

and

$$(\text{prim}(L) \text{ and } \text{mult}(y)) \Rightarrow \text{non } \text{boprim}(Ly)$$

and for some closing remarks.

I shall not go with you in any detail through these proofs and programs. I only mention that I had to replace

$$< \text{exp} > ::= < \text{term} > \mid < \text{exp} >< \text{adop} >< \text{term} >$$

first by

$$< \text{exp} > ::= \{ < \text{term} >< \text{adop} > \} < \text{term} >$$

in order to open the way for a repetitive construct in the body of expsearch. Thereafter I had to replace it by

$$< \text{exp} > ::= < \text{adder} >< \text{term} >$$
$$< \text{adder} > ::= \{ < \text{term} >< \text{adop} > \}$$

because I needed the expression "adder(L)" in my proofs and assertions. The syntax for $< \text{term} >$ and $< \text{prim} >$ were subjected to similar massaging operations.

* * *

So much for the description of my experiment. Let me now try to summarize what seem to be the more relevant aspects of the whole exercise.

1) The routines I designed this time were definitely more beautiful than the ones I had written three years ago. This confirms my experience with the formal treatment of simpler examples, when I usually ended up with more beautiful programs than I had originally in mind.

2) A slight change in the interface describing the reading of the next input character caused a more serious change in the overall specifications chosen for sentsearch: the formal treatment exposed the original interface as a seed of complexity.

3) To treat a program absorbing input L formally as a nondeterminstic program assigning, as it were, a "guessed" value to L is a very useful device, so useful, in fact, that all by itself it is probably a sufficient justification for including nondeterminacy in our formal system. (Independently and in another context, also

C.A.R.Hoare was recently led to treat input in this fashion.)

4) Nearly 11 of the 19 pages don't deal with the programs at all! They are exclusively concerned with exploring the given syntax and proving useful theorems about strings, theorems expressed in terms of predicates derived from the given syntax.

4.1) My earlier treatment of this example took only 7 pages: most of the theorems I proved this time were in the older treatment regarded as "obvious".

4.2) Several patterns of deduction appear in more than one proof; the introduction of a few well-chosen lemmata could probably have condensed somewhat what now took 11 pages.

4.3) The formal treatment of a program requires a formal "theory" about the subject matter of the computations. The development of such a theory may be expected to require the introduction of new concepts that did not occur in the original problem statement.

4.4) In the development of such a theory the choice of notation is crucial. (In this exercise the struggle of developing the theory was mainly the search for an adequate notation; once that had been invented, the development of the theory was fairly straightforward and I don't think that the final document contains more than a single line --at the end, where I was getting tired and lazy-- that could cause a serious reader serious problems.)

5) There is a wide-spread belief that such formal proofs are incredibly long, tedious to write and boring to read, so long, tedious, and boring as a matter of fact, that we need at least a computer to verify them and perhaps even a computer to generate them. To the idea that proofs are so boring that we cannot rely upon them unless they are checked mechanically I have nearly philosophical objections, for I consider mathematical proofs as a reflection of my understanding and "understanding" is something we cannot delegate, neither to another person, nor to a machine. Because such philosophical objections carry no weight in a scientific discussion, I am happy to be able to report that my experiment completely belied the said wide-spread belief.

Since many years I have found that when I write an essay in which a program is developed, the total length of the essay is a decimal order of magnitude greater than the length of the program in which it culminates. The transition to a highly formal treatment has <u>not</u> changed that ratio significantly: it has only replaced the usual handwaving and mostly verbal arguments by more concise, much more explicit and, therefore, more convincing arguments. The belief that formal proofs are longer than informal arguments is not supported by my experiment.

The belief that the writing and reading of such proofs is tedious and boring

has also certainly not been confirmed: it was an exciting challenge to write it and those who have seen it have confirmed that it was fascinating to read, because it all fitted so beautifully --as, of course, in a nice formal proof it should!-- . I am tending to regard the belief that these formal proofs must be long, tedious and boring, as a piece of folklore, even as a harmful --because discouraging-- piece of folklore that we had better try to get rid of. The fact that my formal treatment was in all respects to be preferred above my former, informal treatment has been one of the most encouraging experiences from the whole experiment, and I shall not try to hide the fact that I am getting very, very suspicious of the preachers of the refuted belief: they are mostly engaged on automatic verification or proving systems. By preaching that formal proofs are too boring for human beings they are either trying to create a market for their products and a climate favourable for their funding, or only trying to convince themselves of the significance of their work. The misunderstanding is aggravated by the complicating circumstance that their own activities seem to support their beliefs: I have seen a number of correctness proofs that have been produced by (semi-)mechanized systems, and, indeed, these proofs were appalling!

6) The design consisted of a set of procedures; ignoring the possibility of a recursive call --as would have been the case when the second alternative production for < prim > had been omitted-- they form a strict calling hierarchy of four layers deep. It is worth noticing that all through that calling hierarchy the specification of the procedures is of the same simple nature. The fact that, when we go up the hierarchy, we create in a sense more and more "powerful" machinery is not reflected in greater complication of the treatment, more elaborate interfaces, or what have you. This, too, is a very encouraging observation, as it gives us some clue as to what we might expect when we would undertake a more ambitious experiment with a still less simple example.

Somewhere in his writings --and I regret having forgotten where-- John von Neumann draws attention to what seemed to him a contrast. He remarked that for simple mechanisms it is often easier to describe how they work than what they do, while for more complicated mechanisms it was usually the other way round. The explanation of this phenomenon, however, is quite simple: a mechanism derives its usability in a larger context from the adequacy of its relevant properties and when they are very complicated, they are certainly not adequate, because then the mechanism is certain to introduce confusion and complexity into the context in which it is used.

As a result of this observation I feel that there is a reasonable justification for the expectation that a next more ambitious experiment will just confirm my earlier experiences.

* * *

As you will have noticed I have accepted as some sort of Law of Nature that for the kind of programs I talk about, I accept a documentation ten times as long as the raw code, a Law of Nature that relates how we think to the best of our ability when we program to the best of our ability. Those struggling with the maintenance of programs of, say, 100,000 lines of code, must shudder at the thought of a documentation ten times as bulky, but I am not alarmed at all.

My first remark is that for the kind of programs I am talking about, the actual code is apparently a very compact deposit of our intellectual labours. In view of the various --and considerable!-- costs caused by sheer program length, this compactness should be a reason for joy! But then we cannot complain at the same time about the factor ten! You cannot have your cake and eat it....

My second remark to console the man struggling with the 100,000 lines of code is, admittedly, still a conjecture, but a conjecture for which I have not the slightest indication that it might be wrong. The conjecture is that the actual size of 100,000 lines is less dictated by the task he seeks to solve than by the maximum amount of formal text he thinks he can manage. And my conjecture, therefore, is that by applying more formal techniques, rather than change the total amount of 100,000 lines of documentation, he will reduce the length of the program to 10,000 lines, and that he will do so with a much greater chance of getting his program free of bugs.

* * *

As a result of this exercise I discovered an omission from all computer science curricula that I have been familiar with: we don't try to teach how to invent notations that are efficient in view of one's manipulative needs. And that is amazing, for it seems much less ambitious than, say, trying to teach explicitly how to think effectively. When teaching standard mathematical subjects, they get acquainted with the corresponding standard notations and these are fairly effective; so they have good examples, but that is all! I think that it could help tremendously if students could be made aware of the consequences of various conventions, consequences such as forced repetition, or all information sinking into the subsubsubscripts, etc.

My last remark is added because you may have noticed quantitative concerns from my side, such as worrying about the length of formulae and proofs. This is partly the result of a small study of elegant solutions. The study is not completed yet, but one observation stands out very clearly: the elegant solutions are short.

Appendix.

By way of illustration I include an excerpt from EWD550 "A more formal treatment of a less simple example." After the establishment of formulae (7) through (11) --as numbered in EWD550!-- , i.e. the choice in the case of (7), (8), and (11), and the derivation in the case of (9) and (10):

Rs(S, x, c): bosent(S) and non bosent(Sx) and c = sent(S) (7)

< sent > ::= < exp > ; (8)

< bosent > ::= < sent > | < boexp > (9)

boexp(L) => non sent(L) (10)

Re(E, x, c): boexp(E) and non boexp(Ex) and c = exp(E) (11)

the text continues as follows.

"Designing sentsearch in terms of expsearch means that we would like to have theorems, such that from the truth of a relation of the form Re the truth of relations of the form Rs can be concluded. There are three such theorems.

Theorem 1. (Re(L, x, c) and non c) => Rs(L, x, c)

Proof. Assumed:

0. Re(L, x, c) and non c

Derived:

1. boexp(L) with (11) from 0
2. bosent(L) with (9) from 1
3. c = exp(L) with (11) from 0
4. non c from 0
5. non exp(L) from 3 and 4
6. non sent(Lx) with (8) from 5
7. non boexp(Lx) with (11) from 0
8. non bosent(Lx) with (9) from 6 and 7
9. non sent(L) with (10) from 1
10. c = sent(L) from 4 and 9
11. Rs(L, x, c) with (7) from 2, 8, and 10

(End of Proof of Theorem 1.)"

(End of Appendix)