A.P.I.C. Studies in Data Processing
No. 8

STRUCTURED PROGRAMMING

O.-J. DAHL
Universitet i Oslo,
Matematisk Institut,
Blindern, Oslo, Norway

E. W. DIJKSTRA
Department of Mathematics,
Technological University,
Eindhoven, The Netherlands

C. A. R. HOARE
Department of Computer Science,
The Queen's University of Belfast,
Belfast, Northern Ireland

1972
ACADEMIC PRESS
LONDON AND NEW YORK

8. On Comparing Programs

It is a programmer's everyday experience that for a given problem to be solved by a given algorithm, the program for a given machine is far from uniquely determined. In the course of the design process he has to select between alternatives; once he has a correct program, he will often be called to modify it, for instance because it is felt that an alternative program would be more attractive as far as the demands that the computations make upon the available equipment resources are concerned.

These circumstances have raised the question of the equivalence of programs: given two programs, do they evoke computations establishing the same net effect? After suitable formalisation (of the way in which the programs are given, of the machine that performs the computations evoked by them and of the "net effect" of the computations) this can presumably be made into a well-posed problem appealing to certain mathematical minds. But I do not intend to tackle it in this general form. On the contrary: instead of starting with two arbitrarily given programs (say: independently conceived by two different authors) I am concerned with alternative programs that can be considered as products of the same mind and then the question becomes: how can we conceive (and structure) those two alternative programs so as to ease the job of comparing the two?

I have done many experiments and my basic experience gained by them can be summed up as follows. Two programs evoking computations that establish the same net effect are equivalent *in that sense* and *a priori* not in any other. When we wish to compare programs in order to compare their corresponding computations, the basic experience is that it is impossible (or fruitless, unattractive, or terribly hard or what you wish) to do so when on the level of comparison the sequencing through the two programs differs. To be a little more explicit: it is only attractive to compare two programs and the computations they may possibly evoke, when paired computations can be parsed into a time-succession of actions that can be mapped on each other and the corresponding program texts can be equally parsed into instructions, each corresponding to such an action.

This is a very strong condition.

10. On Program Families

In our previous section we have considered the design of a program for a given task, but in doing so, we have considered our final program as an isolated object, a structure standing all by itself and to be judged on its private merits. Its structure was the result of successive decompositions; the purpose of this structure was to make a program in such a way that its correctness could be proved without undue intellectual labour.

In this section I am going to explain why I prefer to regard a program not so much as an isolated object, but rather as a member of a family of "related programs". In traditional terminology we can think about related programs either as alternative programs for the same task or as similar programs for similar tasks.

Why cannot the programmer confine his attention to the program he has to make and why has he to take into account such a whole family as well? For one thing, it is hard to claim that you know what you are doing unless you can present your act as a deliberate choice out of a possible set of things you could have done as well. But if we want to give due recognition to the difficulties that are specific to the construction of large complicated programs, there is a very practical justification. (And we *have* to recognise these specific difficulties: experience has shown that someone's proven ability to do an excellent job on a given scale is by no means a guarantee that, when faced with a much larger job, he will not make a mess of it.)

Certainly, one of the properties of large programs is that they have to be modified in the course of their life-time. A very common reason is that the program, although logically correct, turns out to evoke unsatisfactory computations (for instance unsatisfactory in one or more quantitative

aspects). A second reason is that, although the program is logically correct and even satisfactorily meeting the original demands, it turns out to be a perfect solution for not quite the right problem; one is faced with a re-statement of the problem and adaptation of the program.

The naïve approach to this situation is that we must be able to modify an existing program (and for this the curious term "program maintenance" has established itself). The task is then viewed as one of text manipulation; as an aside we may recall that the need to do so has been used as an argument in favour of punched cards as against paper tape as an input medium for program texts. The actual modification of a program text, however, is a clerical matter, which can be dealt with in many different ways; my point is that if we have our grip on the program text primarily as on a linear sequence of symbols, the task to establish and to describe what has to be modified tends to become prohibitively difficult when the texts get longer and longer.

If a program has to exist in two different versions, I would rather not regard (the text of) the one program as a modification of (the text of) the other. It would be much more attractive if the two different programs could, in some sense or another, be viewed as, say, different children from a common ancestor, where the ancestor represents a more or less abstract program, embodying what the two versions have in common. Hopefully, this common ancestor can be readily recognised in the (prae-)documentation. The intentions are

(1) that the two versions share their respective correctness proofs as far as possible;

(2) that the two versions share (mechanically) as far as possible the common (or "equal") coding;

(3) that the regions affected by the modification are already well-isolated, a condition which is not met when the transition requires "brain-made" modifications scattered all over the text.

Well, this is a lofty goal. It has been inspired by the potential similarity between the task of program modification and program composition: when a program has been built up to an intermediate stage of refinement, what has then been written down is in fact a suitable "common ancestor" for all possible programs produced by further refinements. It is the similarity between "the decision to be changed" and "the decision still left open": in both cases we are left with what remains when we abstract from such a decision.

There is a second source of inspiration to be found in our experience. In the process of step-wise program composition, proceeding from outside

inwards, going towards progressive refinements, we have in the earlier stages not only postponed deciding how certain things would be done, but we have also postponed committing ourselves as to exactly what had to be done: with progressing refinement, more detail about the actual problem statement has been brought into the picture. (Later examples will show this even more clearly than the problem of the prime table.) As a result, our first levels of refinement are equally applicable for the members of a whole class of problem statements.

In other words, in the step-wise approach it is suggested that even in the case of a well-defined task, certain aspects of the given problem statement are ignored at the beginning. That means that the programmer does not regard the given task as an isolated thing to be done, but is invited to view the task as a member of a whole family; he is invited to make the suitable generalisations of the given problem statement. By successively adding more detail he eventually pins his algorithm down to a solution for the given problem.

All this is well-known, each competent programmer does so all the time. Yet I stress it for a variety of reasons. If the given problem statement is an elaborate affair, i.e. too much to be grasped in a single glance, he *must* approach (and dissect) the problem statement in this way (see the section "On our inability to do much"). Secondly, if the given problem is perfectly defined, it is a wise precaution to anticipate as many future changes in the problem statement as one can foresee and accommodate. This remark is not an invitation to make one's program so "general" that it becomes, say, unacceptably inefficient, as might easily happen, when the generalisations of the problem statement are ill-considered (which might easily happen when they have been dictated by the Sales Department!) But in my experience, even in traditional programming, it is a very worth-while exercise to look for feasible generalisations of conceivable utility, because such considerations may give clear guidance as to how the final program should be structured. But such considerations boil down to ..... conceiving (more or less explicitly) a whole program family!

In an earlier section ("On the reliability of mechanisms.") the need for careful program structuring has been put forward as a consequence of the requirement that program correctness can be proved. In this section we are faced with another reason: program structure should be such as to anticipate its adaptations and modifications. Our program should not only reflect (by structure) our understanding of it, but it should also be clear from its structure what sort of adaptations can be catered for smoothly. Thank goodness, the two requirements go hand in hand.