SETL Newsletter # 150

J.T. Schwartz
May 10, 1975

What Constitutes Progress in Programming?

Donald Knuth has called programming an 'art', and has argued the appropriateness of this designation at some length.[1] In this short essay I should like to argue (though of course terms are not necessarily matters of great consequence) that programming is not an art but a nascent science. The distinction that I see is this: art, though ever changing and fresh, does not and cannot progress, since it lacks any real criterion of progress; but science does progress.

To establish programming as a science is therefore to propose a convincing criterion of progress for it. To this end a comparison with mathematics is enlightening. Mathematics is the search for interesting proofs, and for general frameworks which allow interesting proofs to be found. A proof is defined by its target theorem T, but nonrecursively; even after T is conjectured (which may itself be a significant event) its proof can be arbitrarily difficult to find. Thus the moments of progress in mathematics (typically they are discrete and sharply defined) are (simplifying somewhat) the moments at which proofs are found. Note also that once T is proved, and assuming that T is truly interesting, it will illuminate some broader area, and in particular will ease one's approach to other interesting theorems.

There certainly is a side to programming, namely the invention of algorithms meeting efficiency constraints whose satisfiability is non-obvious, which has just this flavor, and which is therefore as much a science as mathematics. (Knuth is of course one of the main developers of this 'single-algorithm' oriented part of programming science.) The Fast Fourier Transform is no less an invention than the Pythagorean Theorem. But should the other side of programming, namely its integrative side, i.e., the growing collection of techniques used to organise large systems of algorithms into coherently functioning wholes, be considered as an infant science also, or must it remain an art?

I argue that this part of programming is a science also, albeit a science only in its infancy. To see that it is, one must observe that the crucial obstacle to the integration of systems of programs providing very advanced function, which will generally be large systems of programs, is met when their complexity rises above the very finite threshold beyond which the mind can no longer grasp them totally. Those who have had the experience of working with systems of this level of complexity will realise that one's ability to cope with them is quite limited, and always threatens to founder entirely.[2] With the active help of a computer, by assembling multi-person groups (less prone to fatigue than individuals), and by concentrating on one system portion at a time one can cope with such systems.

But even while being successfully developed and maintained they remain elusive and largely inexplicable; in a manner never fully comprehended or controlled, they evolve. In contrast, a system which remains below the threshold critical for full comprehensibility can be designed with assurance and implemented with a firm grasp. Those who have dealt with systems of both sorts will realise that systems of the latter kind can be an order of magnitude easier to deal with than structure which lie beyond the comprehensibility threshold.

Thus programming progresses when schemes which make it possible to realise significant function without overstepping this critical threshold are invented. Each such scheme will address some more or less broad application area, and will provide objects, operations, and also a semantic framework within which these objects and operations can be combined together into large structures, the whole allowing significantly many functions which formerly would have required super-threshold realisations to be written out completely without the critical threshold of complexity being crossed. Proof of the success of such a scheme comes when, by approaching a major application in a way comforming to the rules of the scheme, one finds that it has become comprehensible, though it was not so before. A framework of the kind envisaged is of course a language, and another proof of its success will lie in the fact that this language allows one to speak clearly and directly about important matters which previously could only be depicted in roundabout and clumsy ways.

(In mathematics, major definitions have the same effect.) Note also that the restrictions which such a framework embodies can, if they prevent complexity from rising rapidly,be just as important as the flexibility it provides. [3]

Once such a framework has been invented, and when some process or function has been specified in it, it will generally not be hard, though of course it may be tedious, to take this specification and transcribe it, perhaps to gain efficiency, into some available and appropriate programming language. Because numerous errors are bound to infest any lengthy or complicated process of transcription, it is generally useful to implement languages which realise the framework or something close to it in a polished, succint, and helpful a form as possible. Among, other things, this can call for the development of elaborate program analysis methods, which for example may be used to support rich systems of explicit or implicit declaration, to provide sophisticated diagnostics, or to perform optimisation which the user of a language of very high semantic level is expected to omit. But such development is tool-building rather than fundamental progress. In this sense, I consider that SNOBOL and SIMSCRIPT, for all their lack of polish, embody very significant inventions: SNOBOL the string/pattern algebra and a natural framework for organising operations in that algebra; SIMSCRIPT the event and scheduling notions so helpful for simulation.

Similarly, I would say that the interest of ALGOL 68 lies not in its syntactic polish, but in the way it handles object types and coercions, and in the fact that the kind of systematic approach to declarations which it embodies promises to reduce levels of run-time error very decidedly.

Footnotes:

[1] See Knuth, Computer Programming as an Art. CACM 17, 11 (November 1974), p. 667.

[2] This point is central to Dijkstra's essay Concerning Our Inability To Do Much. p. 1 in Structured Programming, O.J. Dahl *et al*, Academic Press 1972.

See also Schwartz, On Programming, Installment I, Item 1: On the Sources of Difficulty in Programming. Courant Institute of Mathematical Sciences, 1973.

[3] This point lies at the heart of Dijkstra's celebrated note Go-To Considered Harmful, CACM 11, 3(March 1968), and of various of Hoare's interesting comments on programming technique, e.g., Monitors: An Operating System Structuring Concept, CACM 17, (October 1974) p. 261.