Mathematics of Program Construction

Roland Backhouse

October 17, 1995

1 What's It All About?

It took Courant and Robbins [3] more than 500 pages to answer the question "What is Mathematics?". The name "Mathematics of Program Construction" given to the conference first held in 1989 and chaired by Jan van de Snepscheut was chosen with much care but the question exactly *what* it names is as hard to answer as Courant and Robbins' question, if not more so. After all, Mathematics is a discipline that has been in existence for more than 2000 years whereas the Mathematics of Program Construction is not yet a recognised field with recognised paradigms and/or results. The question is in fact "what should constitute the mathematics of program construction and what goals should scientists endeavouring to develop it into a recognised field have?". This paper presents one view on the answer to that question.

1.1 Construction

We begin with "construction" because, of the three terms, "mathematics", "program" and "construction", this is the one that is the most characteristic of the field we are striving to create.

Construction is building: designing and making things that do not exist. The word "construction" was deliberately included in the title of the 1989 conference because, at that time, the sum of "mathematics" and "programming" was commonly associated with program verification. The same misconception still seems to prevail.

It's easy to find examples illustrating the difference between construction and verification. The first time that I can recall having been aware of the difference was when I was taught that the sum of the consecutive numbers $1, 2, \ldots, n$ is $1/2\ n\ (n{+}1)$. It must have been about the same time that I read how Gauss *constructed* this property (actually a particular instance of a more general property [1], but let's not be pedantic) as a young child by

mentally writing down the row of numbers and underneath it the same of row of numbers but in reverse order:

$$\begin{array}{ccccccc} 1 & 2 & \ldots & n-1 & n \\ n & n-1 & \ldots & 2 & 1 \end{array}$$

Observing that the sum of two numbers in the same column is constantly $n+1$ and that there are n such sums, Gauss was thus able to construct the general formula.

Although the construction of this property is easy to explain, the property was introduced to me and my class to illustrate the process of *verifying* formulae by mathematical induction. Particularly disconcerting was that, when the step was taken to considering the sum of the squares or the cubes of the first n numbers (thus $1^2+2^2+\ldots+n^2$ or $1^3+2^3+\ldots+n^3$) there was no question whatsoever of presenting a method of *constructing* the appropriate formula; instead the formulae were fished out of the blue and subjected to *verification* by mathematical induction. My teacher was frank in his admission that he had no idea how to *construct* any of these formulae let alone extend the list to higher powers.

I do not suppose that my experience was atypical. There is a major tendency, in publications of all kinds, to verify rather than construct. A solution is presented and its validity is checked. The problem is that it takes considerably more time and effort to explain the process of construction than it does to verify a solution. Moreover, construction requires a higher degree of abstraction.

1.2 Program Construction

Constructive problem solving is a vast field. Program construction is much narrower but nevertheless pretty vast.

Lest the reader make unwarranted assumptions, it is important to say at the outset that "program construction" is not synonymous with the design of algorithms or "programming in the small". Program construction includes the whole process from problem analysis through design to final product. Programs can vary from large systems incorporating millions of lines of code to small algorithms of less than ten lines.

Of course, it is not the case that the mathematics of program construction can have relevance to all aspects of program construction. Indeed, it probably has relevance to only a few — there are many management and human issues involved in large projects that mathematics does not claim to even attack let alone posit solutions.

The leitmotif of program construction is the tension between precision and concision. By their very nature, computers demand an unprecedented and unnatural degree of precision from their programmers —they are after all just machines whose task it is to mechanically carry out our instructions—. Programmers and users alike, however, demand concision because our brains are just not capable of coping with substantial amounts of detail.

Precision and concision tend to conflict with each other, the one being most often achieved at the expense of the other. The reconciliation of precision with concision is a major intellectual challenge; new it may not be, but intense and vital it certainly is thanks to the electronic computer.

1.3 Mathematics

Concision is evident in systems design as uniformity —the minimisation of *ad hoc* features and the maximisation of utility. Concision is achieved by abstraction, the process of eliminating unnecessary detail typically by generalisation.

Mathematics is the one science that seeks to combine rigour with abstraction, and thus to reconcile precision with concision. The mathematics of program construction, whose subject matter is mathematically-based methodologies for building computer software, aims to improve our intellectual skills in recognising, formulating and exploiting suitable abstractions in the process of designing computer software.

2 Contribution

2.1 Potential

Idealistic goals are one thing, but the demand most usually made is for concrete "results". Mathematicians are often obliged to walk a tightrope. Among themselves they will agree that the most important contribution of mathematics is that indefinable entity called "the mathematical method" and they will take great pleasure in the beauty and, above all, simplicity of a tightly constructed argument; to other scientists and particularly to the man in the street they will emphasise "results", the inherent difficulty of the problems they tackle, the complexity of their solutions and the genius of the solver.

The mathematics of program construction has no such identity crisis. It is not result-oriented but method-oriented. Its goal is to enhance the problem-solving techniques needed to design and build computer software. Its aim is

to achieve the utmost simplicity, so that the method may be emulated by others and not remain the preserve of an elite.

We are not trying to signal the discovery of a panacea; there is no "silver bullet" that can offer a dramatic effect on programming practice.

The areas of mathematics that are of most relevance to programming are algebra and logic. Algebra is particularly relevant because it epitomises usability and reusability, the two most essential characteristics of a computer program. Algebra epitomises usability because it stresses the formulation of simple calculational rules, for example the associativity of multiplication and addition; the rules are *usable* because they are simple and concise. Algebra epitomises reusability because it stresses the identification of abstractions that are common to a variety of mathematical domains, associativity being again a good example; good abstractions are *reusable* because they occur again and again even outside the areas in which they were first identified. Logic is relevant to programming because it is the glue that binds together all the (algebraic) reasoning that is used in program design [4].

This is not to say that every programmer should be subjected to a lengthy apprenticeship in advanced algebra and advanced logic before being allowed to embark on a profession in the software industry. The practising programmer needs to be skilled in using logic and algebra rather than be a logician or an algebraicist. Those whose goal it is to improve the programming method, on the other hand, need to be all three — good programmers, good logicians and good algebraicists!

2.2 The Ideal

Without a shadow of a doubt, the best illustration of the potential benefits of reconciling precision with concision is afforded by the Algol 60 report [5]. The Algol 60 report introduced the so-called Backus-Naur form (BNF) for precisely describing the syntax of a programming language. The simplicity and concision of BNF not only made it much easier for newcomers to learn Algol 60, it made a significant contribution to the structure of programming languages and their compilers.

The Algol 60 report struck a judicious balance between formality and informality based on the state of the art at the time it was written. By so doing it set a standard of communication that is rarely equalled. As our fluency with mathematical reasoning increases we can expect the balance to tip more and more towards formality *without* impeding communication or understanding. That is the ideal for which we are striving.

3 Conclusion

The developed world has great expectations of its scientists and technologists. This in spite of the fact that, according to a number of leading figures, science and technology have relevance to at best one tenth of the world's problems, and, according to a recent survey, only thirteen percent of the population of the U.S. has "a minimally acceptable level of understanding of the process of science".

The expectations and ignorance of society at large are often exploited by the scientists themselves in their rush to win the hearts and minds of those who hold the purse strings; the almost magical development of computer hardware over the last thirty years has led society to expect similar magical developments in computer software. As a result, short term marketing policy driven by a fear of losing the national software industry, rather than the pursuit of knowledge or the development of the intellect, continues to impede the development of the *science* of computing. But the demise of intellectual capital is the most serious threat to a nation's welfare. The folly of expecting magical solutions could not be better expressed than by the words of Frederick P. Brookes [2] when he wrote:

> The first step toward the management of disease was replacement of demon theories and humours theories by the germ theory. That very step, the beginning of hope, in itself dashed all hopes of magical solutions. It told workers that progress would be made stepwise, at great effort, and that a persistent, unremitting care would have to be paid to a discipline of cleanliness. So it is with software engineering today.

And so it is with the mathematics of program construction.

References

[1] Eric Temple Bell. *Men of Mathematics*, chapter Gauss, The Prince of Mathematicians. Simon and Schuster, 1937.

[2] Brooks, Frederick P., Jr. No silver bullet—essence and accident in software engineering. In H.-J. Kugler, editor, *Information Processing 1986, the Proceedings of the IFIP Tenth World Computing Conference*, pages 1069–1076, Amsterdam, 1986. Elsevier Science, B.V.

[3] R. Courant and H. Robbins. *What is Mathematics?* Oxford Univ. Press, London, 1941.

[4] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math.* Springer-Verlag, 1993.

[5] P. (Ed.) Naur. Revised report on the algorithmic language ALGOL 60. *Comm. ACM*, 6:1–20, Also in *The Computer Journal*, 5: 349–67 (1963); *Numerische Mathematik*, 4: 420–52 (1963) 1963.