Mathematics and Programming
A Revolution in the Art of Effective Reasoning

Roland Backhouse*

Inaugural Lecture
24th October, 2001

Abstract

The modern world is highly dependent on the reliable functioning of computer software. The sheer scale of software systems makes their design and implementation a highly demanding intellectual activity. Meeting these demands has inspired a revolution in the way that mathematics, the art of effective reasoning, is conducted and presented. Continued effort is needed in education and research in the mathematical construction of programs, based on the controlled manipulation of mathematical formulae.

At a recent computer expo, Bill Gates reportedly compared the computer industry with the car industry. "If General Motors had kept up with technology like the computer industry has" he is reported to have said, "we would all be driving $25 cars that get 1,000 to the gallon".

"That may be true" was the swift response of the president of General Motors, "but who would want to drive a car that crashes twice a day!".

As you will have guessed, this is just a joke. But the joke is not without foundation. It is true that there have been major, almost incredible advances in computer hardware and software. However, it is also true that computer software is often highly unreliable and liable to crash spontaneously — as it did for Bill Gates during a demonstration he gave of the Windows 98 system at the 1998 COMDEX expo.

*School of Computer Science and Information Technology, University of Nottingham, Nottingham NG8 1BB, England

> "While we're all very dependent on technology, it doesn't always work"
>
> Bill Gates, when Windows 98 crashed during a demonstration he was giving at the Comdex Expo, April 20, 1998.

So-called "computer errors" are frequently reported in the newspapers. For example, on the weekend that I was preparing this lecture, the report you see on the screen appeared in the Guardian. The errors are, of course, not "computer errors" but programming errors made by human beings.

> NatWest was hit by a computer error that affected those who withdrew £20 from cash machines last weekend. They were debited twice.
>
> *The Guardian* October 13 2001

Computer software is often unreliable. But even so, the president of General Motors would probably not dare to complain about crashing computers. In any case it would be very ironic if he did — in the nineteen-fifties General Motors built cars that were liable to spontaneous crashes. The company also succeeded in sabotaging an advertising campaign by the Ford Motor Company to promote increased safety in their cars. (The quotes shown on the screen are taken from Ralph Nader's acclaimed book "Unsafe at Any Speed" and are reported to be statements made by leading figures in General Motors at the time.)

> "Ford sold safety while Chevy sold cars."
>
> "Sales strategies and safety do not mix."
>
> Quotations taken from Ralph Nader "Unsafe at any speed" (1965).

There are other parallels between the car manufacturers of the nineteen-fifties and sixties and software manufacturers of today. Then the car manufacturers tried to put the blame on the drivers, software manufacturers now seem to want to put the blame on the users. During the preparation of this talk, the computer crashed on me on three occasions. On the screen is the message I got when I rebooted.

> Because Windows was not properly shut down, one or more of your disks may have errors on it.
>
> In order to avoid seeing this message again, always shut down your computer by selecting Shut Down from the Start Menu.

Ladies and Gentlemen, I have said enough about the computer industry. That is not my topic. My concern is with developing a science of computing.

> "The hallmark of a science is the avoidance of error"
>
> Robert J. Oppenheimer

Fitness-for-purpose, reliability, dependability. These are the core elements of my agenda. And, fortunately, these concerns have been paramount on the research agenda of some of the most influential university academics since the very beginnings of the development of the science of computing. This has led to a revolution in the way that mathematics is conducted — a quiet revolution, one that has not and is never likely to hit the headlines of the newspapers, but nevertheless an important revolution that will have significant and lasting impact on the way that future computer systems are designed and built. It is about this revolution that I want to talk today.

The main issues in computing are the problems of scale. Computer programs are mathematical formulae, with a precise formal meaning and embodying constructive theorems about the system they implement. But, unlike conventional mathematical formulae, which are compact and manageable, computer programs often extend to tens of thousands and sometimes to millions of lines of code. The huge complexity of computer systems is further compounded by the legacy of unreliable software making so-called "defensive programming" —developing systems that can withstand faults in other components with which they communicate— a necessity. The utmost economy, of expression and of thought, is vital to our ability to master this complexity.

Computer programs embody constructive theorems about the systems they implement. But there are major differences between these theorems and the ones normally studied by mathematicians. For one, they are often shallow and relatively simple. But these are theorems that are applied incredibly often during the execution of a program —millions of times per second— whereas the theorems studied by mathematicians may with luck be applied

a handful of times and sometimes never at all. Even worse the programmer's theorems are applied by an unforgiving machine, rather than an educated and understanding human being, with the effect that even the smallest error is brutally punished, as Bill Gates found out to his cost at the 1998 Comdex expo.

The vast complexity of computer software makes its unreliability understandable, but not excusable. The implementation of reliable computer systems, carrying a guarantee of fitness-for-purpose, imposes major intellectual challenges that can only be met by a science of computing whose hallmark is the avoidance of error.

Equational Logic

In rising to this challenge, computing science has already had a major influence on developments in mathematics. Perhaps the most obvious of those influences is the modern emphasis on formal logic. Nowadays, most, if not all, undergraduate degrees in mathematics and computing science will include some training in formal logic. Thirty-five years ago, when I began a degree in mathematics at Cambridge University, that was not the case. One of the first lectures I attended still stands clear in my mind. The lecturer wrote an upside-down A ($\forall$) on the board and one very brave student had the temerity to ask what it meant. "Oh, don't you know that," said the lecturer. "That means for all. And I will also be using this symbol" —at which point he wrote a backwards E ($\exists$) on the board— "and that means there exists." And that, ladies and gentlemen, was the sum total of my formal education in logic, anno 1966.

Of course, I do not wish to imply that mathematicians have failed to recognise the importance of logic. The line from Hilbert's promotion of formal logic to Turing's brilliant and world-famous paper on the Entscheidungs Problem is well-known. But the needs of computing science has brought formal logic to the fore because the software engineer is a voracious *user* of logic rather than one who studies logic and logical systems.

Computing science's influence on the development of logic is still continuing and impinges on the very fundamentals of the way logic is presented. The computing scientist demands simplicity and conciseness of expression. A good example of this is the modern computing scientists emphasis on

equational reasoning in logic rather than the traditional emphasis on logical implication.

Equality is the most fundamental of mathematical operators if only because of the rule of substitution of equals for equals, the rule for which Leibniz is duly famous.

Equational reasoning, as in for example, school algebra is familiar and straightforward. But equational reasoning is obscured in traditional logic texts because equality of propositions is introduced as "if and only if" —the combination of two operators "if" and "only if"— rather than as an operator in its own right.

It sometimes comes as a surprise to those brought up on "if and only if" to be told that it is just the equality operator, so on this slide I have compared its truth table with the truth table for equality of numbers. They both have the same form: the true entries are along the diagonal and the false entries are off the diagonal. (It also comes as a surprise to many that "exclusive or" is "different from" or the negation of equality, but that is a different story.)

"If and only if", "necessary and sufficient" condition: these phrases make something that is very basic sound very complicated. Compared with logical implication (only if), equality is very simple. And simplicity and economy of thought and expression are the bywords of the practising programmer.

Equality of propositions ("if and only if")

=	t	f
t	t	f
f	f	t

Equality of numbers ("is")

=	0	1	2	3	4
0	t	f	f	f	f
1	f	t	f	f	f
2	f	f	t	f	f
3	f	f	f	t	f
4	f	f	f	f	t

Another surprise for those brought up on logical equality as "if and only if" is that it is associative. Very simple and commonplace examples can be used to illustrate this fact.

Consider the multiplication of two non-zero numbers y and z. The result is positive if y and z have the same sign, that is if the boolean value "y is positive" equals the boolean value "z is positive". The product is negative if they are different, one is positive whilst the other is negative.

> For all non-zero y and z:
> $y \times z$ is positive $=$ (y is positive $=$ z is positive)

Now equality of boolean values (if and only if, if you will) is associative. That is, we immediately have the property that the sign of y times z equals the sign of y exactly when z is positive.

$$(y \times z \text{ is positive} = y \text{ is positive}) = z \text{ is positive}$$

This use of the associativity of boolean equality is a beautiful example of the economy of expression that is made possible by equational logic. But it is an example that often confuses mathematicians and logicians: the confusion arises because there is no way of pronouncing the formula in natural language that allows us to also capture the associativity property. Here is where formal logic takes us beyond the limitations of "natural" modes of reasoning. But also, the fact that the true nature of logical equality is obscured by the use of "if and only if" is perhaps excusable; after all the equality symbol was only introduced in 1557 — which is very recent in the history of mathematics.

> The purpose of logic is not to mimic verbal reasoning but to provide a calculational alternative.
>
> Edsger W. Dijkstra [Dij90]

Construction versus verification

The needs of programmers to find simple but effective ways to specify and reason about their programs has also resulted in major fundamental changes in the way that the reasoning process is recorded and communicated to others.

The programmer's task is to *construct* programs to meet their specifications. Mathematical proof is of course fundamental to establishing the correctness of the decisions made by the programmer but, for the practising programmer, the traditional post-hoc *verification* of mathematical theorems is just not good enough. The process of *construction* is paramount and mathematical design tools must reflect this.

I can illustrate the difference between construction and verification with another recollection I have of my own early mathematical education. When I was at school, in the 6th form, we learnt about proof by induction. We learnt, for example, how to *verify* that the sum of the first n positive numbers is $n(n+1)/2$.

$$\begin{array}{ccccccccc} 1 & + & 2 & + & \ldots & + & n & = & n(n+1)/2 \\ 1^2 & + & 2^2 & + & \ldots & + & n^2 & = & n(n+1)(2n+1)/6 \end{array}$$

We also learnt how to verify that the sum of the squares of the first n positive numbers is $n(n+1)(2n+1)/6$. I remember being fascinated with this process, so much so that I still remember these formulae. My interest was no doubt encouraged because I found proof by induction a very easy process to apply. In those days I was supposedly good at mathematics — we were given a handle that we could turn and I could turn it pretty quickly and score high marks in the exams.

My fascination, however, quickly turned to disillusion. The reason was that I began to think about how to apply proof by induction to other examples. I wondered for example about summing the cubes of the first n numbers

$$1^3 + 2^3 + \ldots + n^3 = ???$$

and summing higher powers:

$$1^{27} + 2^{27} + \ldots + n^{27} = ???$$

But the problem was that I didn't know the answers! I didn't know the formulae to be filled in on the right side of the equations! And to apply proof by induction as I had been taught required one to know the right answer first. (The method I had been taught was to look for a pattern, formulate a conjecture, and then verify the conjecture — in other words, *guess and*

verify. But after many attempts at guesses, I had to admit defeat and give up.)

This was my first encounter with the difference between construction and verification and my first feeling of unease with the way mathematics was conducted. To put it very bluntly, my disillusion was this: at the time, I wasn't really interested in learning facts discovered by Newton, Galois, Gauss, Cauchy or whatever great mathematician of years gone by. What I wanted to know was how *I* could discover new facts and develop new ideas. And the "guess and verify" method did not impress me then and still does not impress me now.

Concrete Mathematics

The emphasis on construction rather than verification permeates a number of texts in computing science. A prominent example is the text "Concrete Mathematics" by Graham, Knuth, and Patashnik [GKP89]. Concrete Mathematics is, according to its authors, "the controlled manipulation of mathematical formulas, using a collection of techniques for solving problems."

Elsewhere they give their goals: "One of the main objectives of this book is to explain how a person can solve recurrences *without* being clairvoyant." The conclusion of this will be that: "Once you, the reader, have learned the material in this book, all you will need is a cool head, a large sheet of paper, and fairly decent handwriting in order to evaluate horrendous-looking sums, to solve complex recurrence relations, and to *discover* subtle patterns in data." The emphasis on the word "discover" is my own but clearly also very important to them. It is not surprising that two of the authors of this text are world-renowned both as mathematicians and computing scientists as the *discovery* of "subtle patterns" is one of the major intellectual demands of computer programming.

An important aspect of enabling construction as opposed to verification is making the rules of the game completely clear. In other words, using formal calculational rules. This is what Graham, Knuth, and Patashnik mean by "the controlled manipulation of mathematical formulas".

An example of the sort of simple, but effective innovations is the formalisation of rules for manipulating quantifiers. I expect most of the audience

will be familiar with the Sigma notation for summing a set of values. This is an example of a quantification where in this case the quantifier is Sigma(Σ), meaning summation. Other elements of the quantification are the *bound variables* (k), the *range* (from 1 to n), and the *term* being summed (k^2).

$$\sum_{k=1}^{n} k^2$$

What Graham, Knuth and Patashnik do in their book is very simple. They begin by stating explicitly the rules for manipulating such summations. The rules are, mostly, very simple — for example, it is permitted to rename the bound variables provided that the new name has not already been used elsewhere in the formula.

$$\sum_{k=1}^{n} k^2 \quad = \quad \sum_{j=1}^{n} j^2$$

Other rules are equally simple, but nevertheless vital. For example, the sum over an empty range is zero.

$$\sum_{k=1}^{0} k^2 \quad = \quad 0$$

By formalising the rules of manipulation we increase understanding and facilitate accurate formal calculation, with enormous benefits for problem solving. This is what they mean by the "controlled manipulation of mathematical formulas".

We can go further than Graham, Knuth and Patashnik's book. Quantifications other than summation abound in computing problems. In mathematics, think of universal and existential quantification and the infimum or supremum of a set of values — but the notation is highly unsystematic and the manipulation rules are rarely formalised.

$$\forall x.p(x)$$

$$inf(x_n)$$

An example of a quantification taken from database technology is a query. The query shown returns the set of all authors who have written books with "Effective Reasoning" in the title.

```
select author: Y
from biblio._ X,
     X.author Y,
     X.title Z
where ‘‘Effective Reasoning’’ in Z
```

Like summation we can recognise in this query a quantifier (`select`), a number of bound variables (`X, Y, Z`), the range of the quantification (books with "Effective reasoning in the title") and the term that is being quantified (`author: Y`).

The added value of a formal study of the "controlled manipulation" of finite summations and other quantifications is that the rules apply just as well to manipulating queries of this nature. So not only do we increase understanding, we are also well equipped to determine better ways of satisfying the query or determining the consequences when the query fails. (Computer crashes often occur because the programmer has failed to take proper account of extreme cases. The supposition is that the users of a database will pose meaningful queries. But many queries are automatically generated —often by unreliable software— and, in any case, human beings are not very reliable either!)

Goal-Directed Constructions

Construction as opposed to verification also has major implications for the way that mathematical theorems are presented. The traditional theorem-followed-by-proof reflects the guess-and-verify style of reasoning which is quite inadequate for the construction of computer programs. Instead a goal-followed-by-construction style of presentation is needed.

I can illustrate the difference between the two by a simple numerical example. On the screen you will see a proof that the square root of 2 plus the square root of seven is greater than the square root of 3 plus the square root of 5. It is a Hilbert-style proof which means that each step in the proof is either self-evident or follows simply from previous lines in the proof. For example, step 3 is self-evident, and step 4 follows from step 3 by simple arithmetic.

As a *verification* of the claimed inequality, this proof is perfectly adequate — it is easily checked either by human beings or by machines. But

0.	if $a>0$ and $b>c>0$ then $a+b>a+c>0$	
1.	if $a>b>0$ then $\sqrt{a}>\sqrt{b}>0$	
2.	$224>9>0$	
3.	$\sqrt{224}>\sqrt{9}>0$	(1 and 2)
4.	$4\sqrt{14}>3>0$	(3 and arithmetic)
5.	$57+4\sqrt{14}>57+3>0$	(0 and 4)
6.	$\sqrt{57+4\sqrt{14}}>\sqrt{57+3}>0$	(1 and 5)
7.	$1+2\sqrt{14}>2\sqrt{15}>0$	(6 and arithmetic)
8.	$8+1+2\sqrt{14}>8+2\sqrt{15}>0$	(0 and 7)
9.	$\sqrt{8+1+2\sqrt{14}}>\sqrt{8+2\sqrt{15}}>0$	(1 and 8)
10.	$\sqrt{2}+\sqrt{7}>\sqrt{3}+\sqrt{5}>0$	(9 and arithmetic)

Figure 1: A Formal Proof of $\sqrt{2}+\sqrt{7}>\sqrt{3}+\sqrt{5}$.

as a construction it is appallingly bad. Note that the proof begins with the statement that 224 is greater than 9. But what have 224 and 9 got to do with 2, 7, 3 and 5? An incredible amount of clairvoyance would appear to be needed to construct this simple proof.

The goal-directed construction, on the other hand, is quite straightforward. The goal is to determine the ordering relation between $\sqrt{2} + \sqrt{7}$ and $\sqrt{3} + \sqrt{5}$.

$$\sqrt{2} + \sqrt{7} \quad X \quad \sqrt{3} + \sqrt{5} \quad .$$

So we name the unknown —here I have indicated it by X— and proceed to calculate it. X may be one of less-than, equals or greater-than, and for each of these we have rules such as if a and b are greater than 0,

$$a^2 \;\; X \;\; b^2 \quad \equiv \quad a \;\; X \;\; b \quad .$$

In this way (see fig. 2)all clairvoyance is eliminated and the calculation becomes what it should be —a straightforward exercise in algebraic manipulation. Moreover, the calculation can now be easily generalised to a computer program that solves the problem in the general case that 2,3, 5 and 7 are replaced by arbitrary integers a, b, c and d. And that is very important.

$$
\begin{array}{ll}
 & \sqrt{2}+\sqrt{7} \;\; X \;\; \sqrt{3}+\sqrt{5} \\
= & \quad \{ \quad \text{squaring is invertible and monotonic with respect to } X \quad \} \\
 & (\sqrt{2}+\sqrt{7})^2 \;\; X \;\; (\sqrt{3}+\sqrt{5})^2 \\
= & \quad \{ \quad \text{arithmetic} \quad \} \\
 & 9 + 2\sqrt{14} \;\; X \;\; 8 + 2\sqrt{15} \\
= & \quad \{ \quad \text{addition is invertible and monotonic with respect to } X \quad \} \\
 & 1 + 2\sqrt{14} \;\; X \;\; 2\sqrt{15} \\
= & \quad \{ \quad \text{squaring is invertible and monotonic with respect to } X \quad \} \\
 & (1 + 2\sqrt{14})^2 \;\; X \;\; (2\sqrt{15})^2 \\
= & \quad \{ \quad \text{arithmetic} \quad \} \\
 & 57 + 4\sqrt{14} \;\; X \;\; 60 \\
= & \quad \{ \quad \text{addition is invertible and monotonic with respect to } X \quad \} \\
 & 4\sqrt{14} \;\; X \;\; 3 \\
= & \quad \{ \quad \text{squaring is invertible and monotonic with respect to } X \quad \} \\
 & 224 \;\; X \;\; 9 \\
= & \quad \{ \quad \text{arithmetic} \quad \} \\
 & X \text{ is ``>''} \quad .
\end{array}
$$

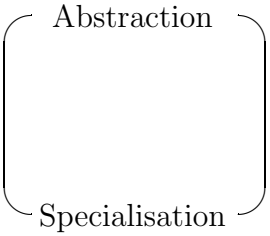
Figure 2: Constructing the relationship between $\sqrt{2}+\sqrt{7}$ and $\sqrt{3}+\sqrt{5}$.

Only by eliminating clairvoyance in this way, can we progress from specific cases to general computer programs.

Unfortunately, the Hilbert-style proof that I so lamented here is very common in mathematical texts —I could show you lots of examples—. The reason is, of course, that verification is so much easier than construction. But a focus on verification rather than construction is ducking the issue. The need to write reliable computer programs is the driving force behind this new, formal calculational style of creating mathematics.

Abstraction and Specialisation

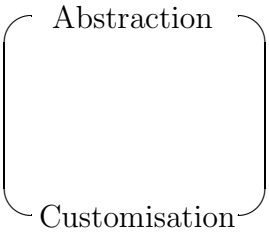
Another important element of programming is the need to be able to *abstract* from the specific to the more general. In science this has long been recognised as the abstraction-specialisation cycle.



Science progresses by abstracting general laws from a variety of observations. These general laws are then applied in specific situations in order to predict new, as yet unknown, properties. These predictions lead to novel applications, yet greater understanding and input for a further round of abstraction followed by specialisation.

The abstraction-specialisation cycle is particularly relevant to the development of the science of computing because the modern digital computer is, above all else, a *general-purpose* device that is used for a dazzling range of tasks. Harnessing this versatility is the core task of software design.

To the software designer the abstraction-specialisation cycle is better known as the abstraction-customisation cycle. Good, commercially viable, software products evolve in a cycle of abstraction and *customisation*.



Abstraction, in this context, is the process of identifying a single, general-purpose product out of a number of independently arising requirements. Customisation is the process of optimising a general-purpose product to meet the special requirements of particular customers. Software manufacturers are involved in a continuous process of abstraction followed by customisation.

The buzz word that is used in this context is "generic": in order to survive, software has to be generic, software libraries have to be generic, software tools have to be generic, software systems have to be generic.

Computing science has facilitated the software designer's task of producing generic software by the continued development of general-purpose, high-level programming languages. Since the first high-level languages were introduced in the early fifties there have been very substantial developments supporting enhanced generality of computer programs.

The progress that has been made can be seen by comparing languages of today with languages of yesterday. All high-level languages have, of course, supported parameterising procedures by values as in, for example, procedures to evaluate the sine or cosine of a real value x. It is the extent to which they support other kinds of parameters that distinguishes languages among each other. In one of the first high-level programming languages, FORTRAN (1957), procedures could be defined, but not used as parameters. In Algol 60 procedures (including functions) were allowed as parameters, but only by name. In neither language could types be named, nor passed as parameter.

Functional languages, of which Haskell is a good example, stand out in the evolution of programming languages because of the high-level of abstraction that is achieved by the combination of higher-order functions and parametric polymorphism. The abstraction mechanisms are also founded on sound mathematical theories that enable us to reason effectively about the behaviour of computer programs.

"The controlled manipulation of mathematical formulas, using a collec-

Date (approx.)	Language	Values (eg integers)	Procedures and functions	Types
$\tilde{1}$957	FORTRAN	yes	no	no
$\tilde{1}$960	Algol 60	yes	yes	no
$\tilde{1}$995	Haskell	yes	yes	yes and no

Figure 3: Parameters

Date	Language	Values (eg integers)	Procedures and functions	Types
$\tilde{1}$957	FORTRAN	yes	yes	no
$\tilde{1}$960	Algol 60	yes	yes	no
$\tilde{1}$995	Haskell	yes	yes	yes

Figure 4: Naming

tion of techniques for solving problems" which is so important to creative, concrete mathematics is possible *and* practical in the development of functional programs. Moreover, the level of generality, when used wisely, considerably eases the burden of proof because the costs are spread over a wide range of applications.

From what I have just said, it would seem that all the programming problems have been solved and we can all go home and sleep contentedly. Unfortunately, that is not the case. The level of genericity still has its limitations and there is an ongoing challenge to find new ways of parameterising programs in a way that is supported by effective reasoning. For example, the so-called "design patterns" [GHJV95] that have recently emerged as a very effective means of designing object-oriented programs have not yet been formulated in terms of precise and concise mathematical abstractions —at least not to my knowledge— nor even can they be captured as statements or declarations in any existing programming language. The use of design patterns is very much *meta* programming, that is programming at a level that is not expressible in the programming language itself. The abstraction-specialisation cycle goes on turning.

Thanks

At this point, I would like to extend my thanks to those who have influenced my own views on mathematics and programming and, in so doing, have fostered my career and helped me to get to where I am now. I am fortunate in that, during my early research career, the dominant leaders in the field —Donald Knuth, Tony Hoare, Edsger Dijkstra, Niklaus Wirth, to name just a few— were scientists who clearly regarded the twin activities of *education and research* as equally important.

I want to stress this point because I fear that the current generation of young researchers is being corrupted by a climate that rewards quantitative measures of research above all else, this climate having been created by an assessment system that separates the functions of research and education. It is not that I am against assessment, it is the lack of recognition of the duality of these activities in the assessment process that I believe is distorting the way that university research is currently conducted. And, unfortunately, this seems to be particularly true in the case of computing science where the commercial potentials and interests are so obviously predominant.

> In a research-oriented university ... the major transfer of information ... to industry does not occur through journal articles and publications; rather it comes about through students who get degrees and then take jobs in industry.
>
> John E. Hopcroft [Hop86]

In my career I have been very fortunate in being able to rub shoulders with two truly great computing scientists — Tony Hoare (now Sir Tony Hoare) and Edsger Dijkstra. I would like to thank Tony Hoare for the encouragement and support he has given me over a number of years, starting at a very early stage in my career in his role as editor of the series of books in which my own two books were published. I hope that I may continue to write books and articles worthy of his high standards of clarity and rigour.

I would also like to thank Edsger Dijkstra for the example he has set in his research and teaching. The phrase "the art of effective reasoning" in the subtitle of my talk is due to Dijkstra — it is his definition of mathematics.

The example that Dijkstra has set and which I am striving to emulate is still not properly appreciated and respected by many computing scientists.

Only just last month I was at a conference at which a prominent computing scientist complained about Dijkstra's "polished" performances, The subject of the conference was software reliability and the argument was, apparently, that "faultless" program design was unnatural and beyond mere mortals and that to pretend that it could be achieved was a distortion of reality.

I find this attitude both saddening and defeatist. To see how saddening it is, let me draw a comparison with the world of sport. In sport, top athletes strive for perfection and are not ashamed to admit it. Even those who know that they will never reach the top still strive to do as well as the top athletes, and again are not ashamed to admit it. What characterises top athletes from the rest is economy of effort, achieved by honing their skills by constant practice. But when it comes to our most precious skill, our ability to reason effectively, it would appear not to be acceptable to strive for perfection, because that is seen as arrogance, and it is not considered acceptable to challenge our students, because that is seen as being elitist.

> "I believe it is fundamentally wrong to teach a science like programming by reinforcing the students' intuition when that intuition is inadequate and misguided. On the contrary, our task is to demonstrate that a first intuition is often wrong and to teach the principles, tools, and techniques that will help overcome and change that intuition! Reinforcing inadequate intuitions just compounds the problem."
>
> David Gries [Gri90]

Conclusion

Ladies and Gentlemen, our lives and livelihoods are now highly dependent on the reliable functioning of computer software. The sheer scale of software systems makes their design and implementation a highly demanding intellectual activity. Meeting these demands has inspired a revolution in the way that mathematics, the art of effective reasoning, is conducted and presented. Continuing effort is needed in education and research in the mathematical construction of programs, based on the controlled manipulation of mathematical formulae. I will strive to promote these activities in every aspect of my work and I look forward to the continued support of the university and my colleagues in achieving my objectives.

Acknowledgements and Bibiliography

Many of the ideas and opinions expressed in this lecture have been heavily influenced by or are directly attributable to others. Where I know the source I give bibliographic details below. In some cases, unfortunately, I no longer know where the idea came from. (Anyone who does is welcome to contact me so that I can include the appropriate reference here in future revisions of this bibliography.) My thanks go to the many friends, colleagues and fellow computing scientists who have helped formed my views, whether or not they are explicitly acknowledged below.

Equational logic, and in particular the exploitation of the associativity of logical equivalence, was introduced by Dijkstra and Scholten [DS90]. A good introductory text is [GS93]. The latter text also introduces a uniform notation for quantification together with rules for their manipulation. (The notation, apart from minor lexical differences, is due to Dijkstra and his colleagues at Eindhoven University of Technology.)

See [Sim95] for further discussion of the abstraction-customisation cycle, particularly with respect to the design of computer software.

For an introduction to generic programming see [BJJM99]. The account of the development of parametrisation mechanisms in programming languages is adapted from the discussion in this paper, and is due to Lambert Meertens.

References

[BJJM99] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming. An introduction. In S.D. Swierstra, editor, *3rd International Summer School on Advanced Functional Programming, Braga, Portugal, 12th-19th September, 1998*, volume LNCS 1608, pages 28–115. Springer Verlag, 1999.

[Dij90] Edsger W. Dijkstra, editor. *Formal Development of Programs and Proofs*, chapter Fillers at the YOP Institute, pages 209–228. The UT Year of Programming Series. Addison-Wesley Publishing Company, 1990.

[DS90] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, Berlin, 1990.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics : a Foundation for Computer Science*. Addison-Wesley Publishing Company, 1989.

[Gri90] David Gries. *Formal Development of Programs and Proofs*, chapter Influences (or Lack Thereof) of Formalism in Teaching Programming and Software Engineering, pages 229–236. The UT Year of Programming Series. Addison-Wesley Publishing Company, 1990.

[GS93] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Springer-Verlag, 1993.

[Hop86] John E. Hopcroft. The impact of robotics on Computer Science. *Communications of the ACM*, 29(6):486–498, 1986.

[Sim95] Charles Simonyi. The death of computer languages, the birth of intentional programming. Proceedings of the 28th Annual International Seminar on the Teaching of Computing Science at University Level, Sponsored by ICL and University of Newcastle upon Tyne, Department of Computing Science, September 1995.