Teaching

Calculation and Discrimination: A MORE EFFECTIVE CURRICULUM

David Gries

There is real concern, and not only on the part of computer scientists, with the lack of rigor and accountability in software engineering.

For example, consider the recently released report [8] by the subcommittee on Investigations and Oversight of the House of Representatives Committee on Science, Space, and Technology, which addresses problems of software system safety, reliability, and quality. This report, in part, criticizes the universities for providing inadequate education for software engineers—both in their discipline and in ethical training related to their discipline. Cherniavsky, in commenting and summarizing this report in an article in *Computing Research News* [4], says the following:

> [. . . there is] a fundamental difference between software engineers and other engineers. Engineers are well trained in the mathematics necessary for good engineering. Software engineers are not trained in the disciplines necessary to assure high-quality software. . . . The problem is not so much not having the mathematics necessary to solve the software problem, but instead having the trained software engineers.

As another example, the Computer Science and Technology Board's recently completed report on the research agenda for software engineering [5] indicates the need for strengthened mathematical foundations in the work force:

> In the absence of a stronger scientific and engineering foundation, complex software systems are often produced by brute force. . . . As software engineers begin to envision systems that require many thousands of person-years, current pragmatic or heuristic approaches begin to appear less adequate to meet application needs. In this environment, software-engineering leaders are beginning to call for more systematic approaches: More mathematics, science, and engineering are needed.

In the face of the growing problems of developing and managing more and more complex software systems, This program, instruction manual, and reference materials are sold "as is" without warranty as to their performance, merchantability, or fitness for any particular purpose. The entire risk as to the results and performance of this program is assumed by you.

However, to the original purchaser only, the publisher warrants the magnetic media on which the program is recorded to be free from defects in materials and faulty workmanship under normal use for a period of ninety days from the date of purchase

FIGURE 1.
A Statement on a Software Product

the report calls for a more rigorous use of mathematical techniques, in the hope that this can help researchers manage and diminish complexity. Promising directions, the report says, include the application of formal methods, which involve mathematical proofs.

One symptom of the problem with software production is the lack of professionalism in the field. Few software products are guaranteed, and many products contain statements like the one in Figure 1. Note that this statement refuses even to refund the price of the software, should it not live up to expectations, yet there is a guarantee for the hardware!

The lack of professionalism is not limited to software firms that develop programs for the PC market. In large corporations, one can find many instances of software written from poorly prepared requirements and specifications, where a more professional engineering practice would have been to rewrite the specification completely before beginning design and development. No professional architect, bridge builder, or car designer would work with specifications of the shoddy nature that one finds in software engineering. Many software engineers lack the judgment to determine whether their task is well defined, or at least the sense of responsibility and confidence to complain when it is not well defined. One hears that software projects are larger and more complex than other classical engineering projects, but that is even more—and not less—reason to be more professional in software engineering.

The maintenance of programs is another area in which lack of rigor, precision, clarity, and professionalism is evident. Many programs are difficult to modify in order to reflect changing specifications only because they are poorly organized, poorly written, and poorly documented.

The problem with software is not limited to the software-engineering profession. As editor of several journals, most notably *Information Processing Letters,* I have read far too many papers submitted by computing scientists that contained poorly presented algorithms, which if published would force each reader to waste far too much time. I have critiqued many papers, showing how the algorithms could be presented more effectively. Generally, the authors have been grateful for the help, and in at least five instances I have been asked to be a coauthor simply because I made an algorithm presentable! In general, computing scientists and engineers show amazingly little ability to present algorithms effectively and are setting appallingly low standards for the next generation to follow.

Moreover, poor presentations of algorithms in texts and lectures cause a great waste of time and effort in courses on data structures, operating systems, compiling, and the like.

In summary, software engineering, computing, and computing education all suffer from a lack of basic mathematical skills that are needed in dealing with algorithmic concepts.

A Common Perception of Formal Methods

The formal techniques that I am discussing involve a *calculational* style of working, in which, at least part of the time, formulas of a calculus are manipulated according to the rules of that calculus. The techniques are not restricted to programming, but can be beneficial in parts of mathematics as well. (Also, they are not the only techniques needed in programming or mathematics.)

Currently, formal techniques and their application in programming are taught too late (if at all) to programmers and software engineers in industry, to graduate students, and to upper-level undergraduates. Since these people do not have the basic skills needed to apply the techniques with any degree of success, attention has to be divided between teaching the basic skills and discussing their advanced applications. Consequently, both topics suffer. To put it bluntly, instructors of graduate software-engineering programs, like that at the Software Engineering Institute in Pittsburgh, are forced to spend time introducing material at the

This article will be presented as the keynote address for the 1991 SIGCUE Symposium in a joint session with the ACM Computer Science Conference on March 7, 1991 in San Antonio, Tex. Gries will receive the annual SIGCUE Award for Outstanding Contributions to Computer Science Education.

Master's degree level that should have been taught at the freshman or even high school level.

A great deal of the problem lies in the typical perception of logic as an object of study. For example, while texts on discrete mathematics for computer science students have a chapter on logic, the material is rarely used in the rest of the text. Hence, the student and the instructor come away with the feeling that the mathematical tools are of academic interest only. They have seen some of the techniques but lack skill in their use and question their applicability. Certainly, most programmers and software engineers feel this way. So much so that they vociferously voice the opinion that their problems are too big and complex to be handled by those formal, mathematical methods. The retort "We know what we want to do, and it's too big a task to formalize" is heard far too often.

Contrast this with scientists in most other fields. Have you ever heard physicists say that their problems are too big and complex to be handled by mathematical techniques? On the contrary, the size and complexity of their problems force them to turn to mathematics for help.

The negative perception of the role of mathematical techniques in programming is not limited to programmers and software engineers. It can be heard in many computer science graduate courses and industrial short courses given by academic faculty. It is passively voiced by the authors of the far-too-many introductory programming texts that teach programming in a clumsy and awkward manner and by every algorithmicist who presents an algorithm in a less-than-effective operational style.

Chandy and Misra, in their book [3] on foundations of parallel programming, have an insightful essay on the interplay of formalism and intuition. Much programming and mathematics is inspired by intuition, they say, and that will continue. Formalism does not supplant intuition; it complements and supports it. Formal reasoning is not merely intuitive argument couched in mathematical notation; indeed, formal reasoning often allows us to take short cuts that have no counterparts in an informal argument. Formal reasoning also provides a degree of rigor and precision that is almost impossible to obtain using intuition alone. On the one hand, Chandra and Misra say, we should not hesitate to rely on intuition to propose programs and theorems; on the other hand, we should not hesitate to dispense with intuition in our proofs.

However, we can only make substantial use of formalisms if we have had proper education and training, and this education and training has been lacking in our undergraduate curricula.

A few years ago, I reviewed a Ph.D. thesis whose author had used a great deal of mathematical notation, but in a rather strange way. As I studied the thesis, it dawned on me that mathematical notation was used *only to abbreviate English.* For example, a theorem would read, "$\forall$ elements $\in$ the set, $\exists$ a value satisfying property P." The proof would be in the same style, with no attempt at using the mathematics to aid in reasoning. When asked about it, the author readily admitted using mathematical notation only for abbreviating and not for helping him reason. It was quite clear that his education was inadequate.

Overcoming the perception that formal methods are not applicable requires a change in how and what we teach, early in the curriculum. We should be giving the students a real skill with formal methods, so that the methods become as ingrained as the techniques learned in elementary school for manipulating arithmetic expressions.

Teaching Calculational Skills

Every high school student is taught to solve word problems, like the following one.

> Mary has twice as many apples as John. Mary eats two. John throws half of his away because they are rotten. Mary still has twice as many apples as John. How many did each have initially?

We solve this problem as follows. We first translate the statement into a formal, mathematical notation, in this case, into two equations. Using M and J to denote the number of apples Mary and John have initially, we write the equations

$$M = 2*J \text{ and } M - 2 = 2*(J/2).$$

We then solve these equations, using methods that have been taught in class. In this case, we substitute $2*J$ for M in the second equation, yielding

$$2*J - 2 = 2*(J/2),$$

and then solve for J, yielding $J = 2$. Substituting 2 for J in the first equation yields M: $M = 4$.

The next step is to check the answers. We substitute the answers $M = 4$ and $J = 2$ in the second equation and check to see if it is true:

$$\begin{aligned} & 4 - 2 = 2*(2/2) \\ = {} & 2 = 2*1 \\ = {} & \mathit{true} \end{aligned}$$

If an error is found while checking the answer, we go through the calculations performed earlier to determine where a mistake was made.

In summary, part of the mathematical method that is taught in high school goes as follows:

> **Method.** Formalize the problem; solve the problem using known techniques; check the solution; and if the solution is

Theorem. Well foundedness and the validity of mathematical induction are equivalent.

Proof. We have

$$
\begin{array}{ll}
 & (3) \\
= & \langle \text{Definition} \rangle \\
 & \neg empty\ (S) \equiv (\exists\, y :: y \in S \wedge (\forall x < y : x \notin S)) \\
= & \langle \text{Complement both sides, use Negation and De Morgan} \rangle \\
 & empty\ (S) \equiv (\forall\, y :: \notin S \vee (\exists\, x : x < y : x \in S)) \\
= & \langle \text{Define a predicate } P : P.x \equiv (x \notin S)\ ; \\
 & \quad \text{replace occurences of } S \text{ by } P \rangle \\
 & (\forall x : x \in U : P.x) \equiv (\forall\, y : y \in U : P.y \vee (\exists\, (x : x < y : \neg P.x)) \\
= & \langle \text{The above formula is 5} \rangle \\
 & (5)
\end{array}
$$

Since one can argue that for every set S there exists a predicate $P.x \equiv (x \notin S)$, and for every predicate P one can define the corresponding set S, we have proved the theorem.

FIGURE 2.
Proof of Equivalence of Well Foundedness and Math Induction

Theorem. Composition of binary relations is associative: $\rho \circ (\sigma \circ \theta) = (\rho \circ \sigma) \circ \theta$.

Proof. Let $(a, d) \in \rho \circ (\sigma \circ \theta)$. Then there is a b such that $(a, b) \in \rho$ and $(b, d) \in \sigma \circ \theta$. This means that there exists c such that $(b, c) \in \sigma$ and $(c, d) \in \theta$. Therefore, we have $(a, c) \in \rho \circ \sigma$, which implies $(a, d) \in (\rho \circ \sigma) \circ \theta$. This shows that

(6) $\rho \circ (\sigma \circ \theta) \subset (\rho \circ \sigma) \circ \theta$.

Conversely, let $(a, d) \in (\rho \circ \sigma) \circ \theta$. There is a c such that $(a, c) \in \rho \circ \sigma$ and $(c, d) \in \theta$. This implies the existence of a b for which $(a, b) \in \rho$ and $(b, c) \in \sigma$. For this b we have $(b, d) \in \theta$, which gives $(a, d) \in \rho \circ (\sigma \circ \theta)$. We have proven the reverse inclusion

(7) $(\rho \circ \sigma) \circ \theta \subset \rho \circ (\sigma \circ \theta)$.

which gives the associativity of relation composition.

FIGURE 3.
A Mathematician's Proof of Associativity of Relation Composition

wrong, determine where a mistake was made in formalizing or solving the problem.

Now, consider the following related problem.

Mary has an even number of apples. Twice the number of apples that Mary has plus the number of apples that John has is some (unknown) constant C. Suppose Mary throws half her apples away. What should be done with John's apples so that twice the number of apples that Mary has plus the number of apples that John has is still C?

This kind of problem occurs fairly frequently in programming. For example, the body of a loop typically makes progress toward termination (throw half of Mary's apples away), and some other statements are needed to maintain a loop invariant (what should be done with John's apples?). The problem can be formalized as the problem of finding an expression E that makes the following Hoare triple valid:

$$(1)\quad \begin{array}{l}\{even(M) \wedge 2*M+J=C\}\\ M, J := M \textbf{ div } 2, E\\ \{2*M+J=C\}.\end{array}$$

Instead of using the general method for solving this problem, most computing scientists would guess the answer, test it by running it on a computer or hand simulating it, and, if a mistake were detected, would guess another answer. There would be no formalization, no calculation, and, upon finding an error, no attempt to determine the mistake made during the calculation.

This, we believe, is at the heart of the problem in software engineering. There is no attempt to teach methods for formalizing, for solving by calculation, and for checking calculations. The field relies far too much on intuition and guessing.

Problem (1) can actually be solved quite simply. It is equivalent to solving for E in

$$(2)\quad \begin{array}{l}even(M) \wedge 2*M+J=C \Rightarrow\\ wp(\text{“}M, J := M \textbf{ div } 2, E\text{”},\\ \quad 2*M+J=C),\end{array}$$

which can be solved by setting aside the antecedent and manipulating the consequent:

$$\begin{array}{ll} & wp(\text{“}M, J := M \textbf{ div } 2, E\text{”},\\ & \quad 2*M+J=C)\\ = & \langle \text{Def. of} := \text{and textual subst.}\rangle\\ & 2*(M \textbf{ div } 2)+E=C\\ = & \langle \text{Use antecedent to replace } C\rangle\\ & 2*(M \textbf{ div } 2)+E=2*M+J\\ = & \langle \text{Solve for } E\text{, note } M \text{ is even}\rangle\\ & E=M+J\end{array}$$

This example is only the tip of the tip of the iceberg with regard to calculation in programming. Many more examples could be given to show the use of formalizing and calculating, dealing with assignments, loops, recursive functions, and the like.

Here is another example of the use of calculations, due to Dijkstra, which deals with mathematical induction. Generally speaking, students are taught how to perform mathematical induction over the natural numbers. They are not taught *why* it works, and they are not taught how it generalizes to other sets and relations besides the natural numbers and operation $<$.

One can give the students a far better feel for mathematical induction, as well as additional education in formal manipulation, by proving to them that the validity of the principle of mathematical induction over a set U and relation $<$ is equivalent to the pair $(U, <)$ being well founded.

$(U, <)$ is well founded means that every nonempty subset of U contains a minimal element (according to $<$). Using S to denote an arbitrary subset of U, we write this formally as

$$(3)\quad \begin{array}{l}\neg empty(S) \equiv\\ (\exists y :: y \in S \wedge (\forall x : x < y : x \notin S))\end{array}$$

On the other hand, mathematical induction can be formalized as follows (P is a boolean function, or predicate, of one argument, and $P.x$ denotes its application to x):

$$(4)\quad \begin{array}{l}(\forall x :: P.x) \equiv\\ (\forall y :: (\forall x : x < y : P.x) \Rightarrow P.y),\end{array}$$

which, by the laws of implication and De Morgan, is equivalent to

$$(5)\quad \begin{array}{l}(\forall x :: P.x) \equiv\\ (\forall y :: P.y \vee (\exists x : x < y : \neg P.x))\end{array}$$

In Figure 2, we prove, using a calculational style, that well foundedness and the principle of mathematical induction are equivalent. I can attest to the fact that this proof is well within the grasp of junior computer science majors, so much so that they can repeat it on a test. Further, my experience leads me to believe that, with proper education, freshmen will have little difficulty mastering it.

Consider another example, taken from a draft of a text on discrete mathematics, written for computer scientists by mathematicians. Figure 3 is a proof, from the text, that the composition of binary relations is associative. Note that the proof is given basically in English and that it requires *two* proofs, the so-called “if” and “only if” parts.

A calculational proof of the same theorem is given in Figure 4. It is shorter, and it shows directly the equivalence of $\rho \circ (\sigma \circ \theta)$ and $(\rho \circ \sigma) \circ \theta$. It is easier to internalize, since it follows a form that is common to many proofs of properties: replace a notation by its definition, manipulate, and reintroduce the notation.

In showing these examples of a calculational style of proof or development, I am attempting to convince the reader that the style has broad application and that it results

Theorem. Composition of binary relations is associative: $\rho \circ (\sigma \circ \theta) = (\rho \circ \sigma) \circ \theta$.

Proof. We show that (a, d) is in $\rho \circ (\sigma \circ \theta)$ exactly when it is in $(\rho \circ \sigma) \circ \theta$.

$$
\begin{array}{ll}
 & (a, d) \in \rho \circ (\sigma \circ \theta) \\
= & \quad \langle \text{Definition of } \rho \circ (\sigma \circ \theta) \rangle \\
 & (a, b) \in \rho \text{ and } (b, d) \in \sigma \circ \theta \qquad \text{for some } b \\
= & \quad \langle \text{Definition of } \sigma \circ \theta \rangle \\
 & (a, b) \in \rho \text{ and } (b, c) \in \sigma \text{ and } (c, d) \in \theta \qquad \text{for some } b, c \\
= & \quad \langle \text{Definition of } \rho \circ \sigma \rangle \\
 & (a, c) \in \rho \circ \sigma \text{ and } (c, d) \in \theta \qquad \text{for some } c \\
= & \quad \langle \text{Definition of } (\rho \circ \sigma) \circ \theta \rangle \\
 & (a, d) \in (\rho \circ \sigma) \circ \theta
\end{array}
$$

FIGURE 4.
A Calculational Proof of Associativity of Relation Composition

in crisper, shorter, and more precise work. I am not advocating the formal proof of correctness of all programs. I am simply arguing that acquiring calculational skill can produce a marked change in one's perception and use of formal methods.

In developing a calculational skill, one learns that formalization can lead to crisper and more precise descriptions. One learns that the shape of the formalization can itself lend insight into developing a solution. One acquires the urge to clarify and simplify, to seek the right notation in which to express a problem. One acquires a frame of mind that encourages precision and rigor. This frame of mind can have a strikingly beneficial effect on whatever work one does later as a professional in computing.

Teaching Discrimination

Generally speaking, our text books in computing, and hence our courses, teach facts. We teach a programming language. We teach sets, relations, functions, graphs, logic, Turing machines, automata, and formal languages. We teach a few data structures, compiler construction, operating systems, and so forth.

In few places in the undergraduate curriculum do we discuss judgment and discrimination. For example, rarely do we compare the advantages and disadvantages of two styles, or several different methods for performing a task; or introduce different notations and discuss the contexts in which each is better, the reasons for their existence, and their history; or ask students to compare two proofs. Rarely are formal and informal techniques for the same problem juxtaposed.

Consequently, many students think they have learned the way the science is, has to be, and will be in the future. They have not learned that science is a living thing, which changes and grows. They have not learned to question, to think for themselves, to discriminate.

Some students go on to graduate school and, through research, begin to think as scientists or engineers. The majority, however, do not, and the computing profession is poorer for it. We end up with professionals who are unable to make technical decisions on technical grounds. This is unfortunate, because judging and choosing based on technical merit seem to be important in computing, with its many different languages and styles, especially in software engineering.

Thus in the introductory courses, I would place more emphasis on style, taste, making choices based on technical reasons, and comparing advantages and disadvantages. The following are examples of what I am referring to:

Three ways of proving an implication $X \Rightarrow Y$ are (0) transform it to *true* using equivalence transformations; (1) assume X true and, using its conjuncts as new axioms, transform Y to *true*; and (2) assume Y false and prove X false. A course may look at all three methods, but rarely is there any real discussion and comparison of them. The students are simply shown the three methods, given a *different* example of each, and are expected automatically to be able to choose the appropriate one from then on.

Consider the problem of proving $\neg P \equiv P \equiv false$, given various axioms and theorems, including the following three:

Axiom 0 $(\neg P \equiv P) \equiv \neg(P \equiv P)$

Axiom 1 $P \equiv P \equiv true$

Axiom 2 $\neg true \equiv false$

Below are two proofs done by students in a course of mine, in which the majority of the students already had B.S. degrees in computer science. The lefthand proof was the predominant one, even though it is longer—it requires the copying of "$\equiv$ *false*" on every formula except the last.

$$\begin{array}{ll@{\qquad}ll}
 & \neg P \equiv P \equiv false & & \neg P \equiv P \\
= & \langle \text{Axiom 0} \rangle & = & \langle \text{Axiom 0} \rangle \\
 & \neg(P \equiv P) \equiv false & & \neg(P \equiv P) \\
= & \langle \text{Axiom 1} \rangle & = & \langle \text{Axiom 1} \rangle \\
 & \neg true \equiv false & & \neg true \\
= & \langle \text{Axiom 2} \rangle & = & \langle \text{Axiom 2} \rangle \\
 & true & & false
\end{array}$$

Do you think this example is too trivial to dwell on? If we do not teach students to search for the simplest and shortest solutions on small problems, how will they learn to do it on larger ones? Through many exercises, discussions, and comparisons, we can get students to see that a choice can make a big difference, and that they should seek the best choice in each situation.

The conventional mathematical notation for function application is $f(x, y)$. We also use infix notation $x f y$ (for some functions), postfix notation $x y f$, the notation $(f x y)$ used in some functional languages, the notation of currying, and a newer notation $f.x.y$, with which some computing scientists are experimenting. These notations could be discussed and compared, and each one could be used in the context in which it is most appropriate.

For the ancient Greeks, numbering began with 2. In the modern world, most people think the numbers start with 1. However, there are good technical reasons for beginning with 0, and these can be discussed in detail.

In Figure 2, we provided a calculational proof of equivalence of the validity of mathematical induction over $(U, <)$ and well foundedness of $(U, <)$. Some mathematicians and computer scientists would rather see a more English-style proof, with more "intuition." Such a proof, done by a colleague of mine, is given in Figure 5. In discussing the differences in the two proofs, the students will begin to develop their own sense of judgment and discrimination. (I am amazed at how many people like the proof in Figure 5 better, but they never have an answer to my argument that the proof in Figure 2 is shorter and, more importantly, *far* easier to internalize and then to repeat to others. Basically, one needs to know the definitions of well foundedness and mathematical induction and to be able to translate these definitions into logical formulae. Thereafter, simple manipulations are used to translate one into the other, and the shapes of the formulae help tremendously in this translation. Heuristics can be taught that help one gain skill in such manipulations.)

Compare two formulas for expressing mathematical induction: (9) in Figure 5 and (4). One uses an implication and the other an equivalence. Why the difference? Which is better? Does it matter? Which of the two proofs in Figures 2 and 5 proof is shorter? Which is easier to internalize, so that one can repeat it without having to read it again? Why is one longer than the other?

There will be other situations in which the better technique or method is not so clear, and discussions with students will begin to build discrimination based on technical reasons.

One should also illustrate from time to time how our experiences and habits with syntactic formalisms can hurt or help. For example, is the following, in which x, y, and z are integers, true or false? After determining the answer, try it out on your friends; see how long it takes them to solve this problem and what techniques they use in doing so.

$$x > z \Rightarrow x > y \vee y > z$$

As another example, consider the following problem, due to Wim Feijen. Let x **max** y denote the maximum of x and y. Is the following true or false, and how do you prove it?

(11) $x + y \geq x \;\mathbf{max}\; y \equiv x \geq 0 \wedge y \geq 0$

Before looking at the solution given below, try to solve this problem yourself. Be conscious of the techniques you apply.

Often, one attempts to devise a proof by looking at examples, or by performing a case analysis of one sort or another. A better approach may be to depend on a formal definition of **max**, one that allows for calculation. We can define **max** by

$$z \geq x \;\mathbf{max}\; y \;\equiv\; z \geq x \;\wedge\; z \geq y$$

for all z. Then we manipulate the LHS of (11) as follows.

$$\begin{array}{ll}
 & x + y \geq x \;\mathbf{max}\; y \\
= & \langle \text{Definition of } \mathbf{max} \rangle \\
 & x + y \geq x \;\wedge\; x + y \geq y \\
= & \langle \text{Arithmetic} \rangle \\
 & y \geq 0 \;\wedge\; x \geq 0
\end{array}$$

Thus, by relying on formal definitions and calculation, we see in an extremely simple manner that (11) is true.

Overhauling the Beginning of the CS Major

In many undergraduate computer science programs, a second programming course (the one beyond the remedial first course) is followed by a course in discrete mathematics. The second programming course teaches analysis of algorithms, recursion, abstract data types, a bit about correctness proofs, abstraction and design, and the like, sometimes using a functional approach [1, 2]. The discrete-math course teaches logic, relations, sets, formal languages, automata theory, combinatorics, and graph theory. Much of the theory taught in discrete mathematics could be applied in the programming course, but the order in which the material is presented conveys the opposite impression! Many students justifiably question the usefulness of the discrete mathematics courses they have to take. Rarely does either course teach skill in formal manipulation or attempt to convey a sense of judgment and discrimination.

I suggest merging the contents of the two courses into a two-semester course and, at each stage, teaching theory and then putting it into practice. My choice for the first topic would be five to six weeks' worth of mathematical logic, taught so that the students acquire a *skill* in calculation. Completeness, nonstandard logics, and so forth would not be taught at this time; instead, the emphasis would be on the acquisition of skill in formal manipulation. This topic would form the basis for a calculational style of thinking that would permeate the whole two-course sequence. I would rely on references like [6] and [7], which place importance on the form or shape of mathematical arguments in developing heuristics and techniques for shortening and simplifying proofs. They deal with method, rather than simply with facts.

Thereafter, the course could teach the conventional topics, but in a way that would rely on the calculational skill just acquired. The proof of equivalence of mathematical induction and well-foundedness when dealing with mathematical induction is an example. Also, since the students would have a thorough knowledge of logic, it would be far

FIGURE 5.
Alternative Proof of Equivalence of Induction and Well Foundedness

Definition (Decreasing finite-chain property). Consider function $DC\ F.y$ defined by

(8) $DC\ F.y \equiv$ 'Every decreasing chain beginning with y is finite'.

$(U, <)$ has the decreasing finite-chain property iff $DC\ F.y$ holds for all y in U.

Lemma. $(U, <)$ is well founded iff the finite-chain property holds.
Proof. The proof of this lemma is trivial.

Theorem. $(U, <)$ admits induction iff if it is well founded.
Proof. We prove the two directions separately. First, we assume $(U, <)$ is well founded and prove.

(9) $(\forall(y :: P.y \vee \exists(x : x < y : \neg P.x)) \Rightarrow (\forall(x :: P.x)$.

Let P be a predicate on U that satisfies the antecedent of the implication in 9. Consider the set S defined as $\{x \mid \neg P.x\}$. Establishing that S is empty will prove direction of the theorem.

Suppose that S is non-empty. Since $(U, <)$ is well founded, S has a minimal element u (say). We show a contradiction, which proves that the assumption that S is not empty is false. Since u is a minimal element of S, every element of U that is less than u cannot be in S; i.e. every such element satisfies P. Then, by our assumption about P, it follows that u satisfies P. In other words, u is not in S, which is the desired contradiction.

For the other direction, we assume that $(U, <)$ admits induction (i.e. 9 holds for all P) and show that it is well founded. We assert (without formal proof) that the following is a tautology: either $DC\ F.y$ holds or there exists an x such that $x < y$ and $\neg DC\ F.x$ holds:

(10) $\forall(y :: DC\ F.y \vee \exists(x : x < y : \neg DC\ F.x))$

Using the induction principle, we conclude by mathematical induction that $\forall(x :: DC\ F.x)$ holds. By the lemma, 17 $(U, <)$ is well founded.

easier to teach about specifications of a program, the theory of correctness of programs, and the like, and then to put it all into practice.

In such a two-semester course, I would teach both functional and imperative algorithmic styles, illustrating the advantages and disadvantages of each and pointing out similarities in developmental methods for the two. The functional style would probably come first, for it is a simple step from teaching mathematical induction to talking about and manipulating recursive functions. The imperative approach would enter the picture when talking about economy of space and time.

One of the key notions for computer science is *abstraction.* Abstraction seems to be more important to us than to mathematicians—at least we talk about it more. The functional approach seems more appropriate for conveying a sense of abstraction, for building larger program units out of smaller ones. This, however, may be due more to the status of our current understanding of programming notations than to anything else.

There is the age-old problem of giving the students more experience with "real" languages and programming assignments. Where it makes sense, applications of theory should be backed up by computer-based assignments. However, we should not let the problems of our implemented languages, with all their idiosyncracies, dictate the concepts and their applications that we teach. In fact, it would not be so bad if the students began to feel frustrated with some of the languages and implementations that they are forced to use. Exposing the students to several different implemented programming notations within the two-semester course would help them to see the value of knowing several notations and the contexts in which each is useful.

One does have to be careful with the students' time. Too many instructors are insensitive to the fact that students take several courses and have limited time for each. These instructors glory in giving a difficult and time-consuming course and pride themselves on the number of students who drop it. Programming assignments especially are often unreasonable: instructors expect a high level of performance from the students, but do not provide them with the skill needed to complete the assignment effectively (a skill that the instructors often do not have themselves, because of their own poor education).

This macho attitude endears the student neither to the instructor nor to the content of the course. It serves little purpose except to falsely boost the ego of the instructor. Instead, the aim should be to structure our teaching and homework so that the students can learn the maximum amount in the minimum amount of time.

On Method and Design

These days, the notion of an algorithm, as well as some skill in programming, is important in almost every scientific and engineering field. More and more research and practice deals with implementing ideas on the computer and requires the presentation of algorithms.

The computer science viewpoint has also affected some fields of mathematics. New research ideas have sprung from computer scientists' use of logic. Constructive logic has become an area of hot activity, due to its use in extracting programs from proofs. Computational complexity has excited many a mathematician. Category theory is finding applications in computing. All sorts of mathematical concepts are becoming more and more important in applications such as graphics and robotics, requiring mathematicians and computer scientists to work side by side.

In this context, it appears to me that the proposed two-semester course, which includes programming as well as mathematics from a computer-science viewpoint, would be of interest to both engineers and mathematicians.

However, there are even stronger reasons for students from mathematics and engineering to take the proposed course. Let us consider the mathematics student first.

It was mathematician Morris Kline who said, "More than anything else, mathematics is Method." Yet, few courses in mathematics attempt to teach method, and many mathematicians do not even think it can be taught. Rather, they think that method is something one learns in a rather unconscious fashion over the years. (Polya, of course, was interested in method and wrote several illuminating books on the subject.)

I believe that one *can* furnish the student with some idea of method quite early in the game and that this may give the student more of a sense of appreciation for mathematics and how it is done. The simple heuristics that we can teach concerning syntactic calculations in proofs and programs, along with a sense of discrimination, can be of tremendous help to the student. (Of course, we should not convey the idea that all proofs arise simply out of following a few heuristics.)

Computing, like most fields that have been shown to be amenable to mathematical treatment, has benefited greatly from that mathematical treatment. And, like those fields, computing has repayed the debt by enriching mathematics with new areas of concern and new problems to tackle. But another kind of enrichment stands out in my mind: the emphasis on method that is associated with the calculational approach to proofs and programs. The calculational approach is not really new, of course, and many mathematical proofs have been accomplished using it. What is new is the sense of its pervasiveness and its ability to simplify, which comes by treating the propositional and predicate calculi as calculational systems and using them throughout other areas of mathematics, wherever appropriate.

Now let us consider the proposed course as it relates to engineering. The main activity that is supposed to separate engineering from other fields is *design*: the actual activity of preparing plans for some object (a bridge, radio, electronic circuit, whatever). Within the engineering curriculum, design is viewed as a pervasive, pragmatic activity. During design, the student is particularly encouraged to learn to deal with issues of structure and to evaluate designs with regard to different measures, such as cost, time, complexity, reliability, and ease of production.

In this regard, I believe that the two-course sequence that I am proposing could be an important part of the engineering curriculum. First, it attempts to teach something about the design of (some form of) proofs, giving the students heuristics for their construction and a few ways of comparing them (e.g. with regard to simplicity, length, structure). It helps make the student more aware of choices and the need for discrimination. Second, it attempts to teach about the design of programs, using principles and heuristics for algorithmic development that arise out of a theory of program correctness. The need for rigorous specifications before designing a program becomes clear. The use of different notations, each with its suitable domain of discourse, is discussed. The use of abstraction to help structure programs, as well as discussions of notations for expressing structuring, is expounded upon. Comparison of algorithms with regard to time and space requirements, structure, simplicity, and the like is an important activity. Laboratories, in which the students receive practical experience, are an integral part of the course.

Thus, aside from the obvious benefit of more (and useful) mathematical formalism in the engineering curriculum, the second benefit would come from the notion of design based on theoretical principles, which is so evident in the proposed course.

The Effect on More Advanced Courses

Providing a solid mathematical foundation and a skill in manipulation can have a profound effect on later courses. The presentation of almost *any* algorithm becomes easier when the instructor and student share a common basis for the specification and presentation of programs in a rigorous and calculational manner—rather than the ineffective operational approach that is so prevalent today. The idea behind an algorithm can be conveyed more effectively and in less time, and often it is the idea rather than the whole algorithm that counts. Thus, the increased understanding of the students should allow us to cover more material and to compress some of the courses in our burgeoning curricula.

Courses that would be most directly affected by the proposed change in the introductory course are assembly language programming and machine architecture, data structures, algorithms, computational complexity, compiler construction, and operating systems, since they deal most directly with the development and presentation of algorithms.

Conclusion

Software engineering, and to some extent the rest of computing, suffers from a lack of rigor and professionalism, which stems partly from the belief that formal methods in algorithmic analysis, development, and presentation are useless. As long as computing is taught in a manner that conveys the impression that formal methods are useless, students will believe that formal methods are useless. The calculational methods that have been developed in the past 15 years offer hope that the situation can be changed.

Thus far, there has been little attempt to teach this material to freshman in the United States; therefore at this point my opinion that the suggested changes will help is more a matter of faith than fact. That faith, however, is based on the solid experiences I and others have had in using the calculational style in our own programming, in the presentation of programs, and in the teaching of programming.

Calculational techniques deserve to be given a fair chance, especially since nothing else has appeared on the horizon to solve the ills of the profession. Let us all learn more about calculational techniques and gain skill with them; and let us begin to teach computing using them. Then, the 1990s may see the drastic revisions of our texts and lower-level courses that are needed to effect the change. ■

References

1. Abelson, H., and Sussman, G.J. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Mass., 1985.
2. Bird, R., and Wadler, P. *Introduction to Functional Programming.* Prentice Hall, N.Y., 1988.
3. Chandy, K.M., and Misra, J. *Parallel Program Design.* Addison Wesley, Menlo Park, 1988.
4. Cherniavsky, J.C. Software failures attract congressional attention. *Comput. Res. News 2,* 1 (Jan. 1990), 4–5.
5. Computer Science and Technology Board Report. Scaling up: A research agenda for software engineering. National Academy Press. Excerpted in *Commun. ACM 33,* 3 (Mar. 1990), 281–293.
6. Dijkstra, E.W., and Scholten, C.S. *Predicate Calculus and Program Semantics.* Springer Verlag, N.Y., 1990.
7. Gasteren van, A.J.M. On the shape of mathematical arguments. Ph.D. dissertation, Technical University Eindhoven, Eindhoven, The Netherlands, 1988.
8. GPO 052-070-06604-1. Bugs in the

program—problems in federal government computer software development and regulation, 1989. Superintendent of Documents; Government Printing Office; Washington, D.C. 20402.

9. Gries, D. *The Science of Programming.* Springer Verlag, N.Y., 1981.
10. Hoogerwoord, R. The design of functional programs: a calculational approach. Ph.D. dissertation. Technical University Eindhoven, Eindhoven, The Netherlands, December 1989.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.10 [**Software Engineering**]: Design—*methodologies, representation;* F.4 [**Theory of Computation**]: Mathematical Logic and Formal Languages; G.2.0 [**Mathematics of Computing**]: Discrete Mathematics—General; K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms: Verification

Additional Key Words and Phrases: Accountability, rigor

About the Author

DAVID GRIES has been a faculty member of the computer science department at Cornell University since 1969 and its chair in 1982–87. He has a doctorate in mathematics from the Munich Institute of Technology, and his research interests revolve around programming methodology. **Author's Present Address:** Computer Science Department, Upson Hall, Cornell University, Ithaca, NY 14853-7501.