environment, but it is an effective and well-designed piece of software. Even a version of CC without graphics would be a desirable package.

CC is copyrighted by David Meredith and distributed by

Mathematics Application Group
Department of Mathematics
San Francisco State University
1600 Holloway Avenue
San Francisco, CA 94132

CC is shareware and the authors of the program suggest a $25 donation to the Applied Mathematics Fund at SFSU. Further information can be obtained by calling (415)338-2199.

Mathematical Proofs of Computer System Correctness

*Jon Barwise**

In the early years of the century, there was an exciting, if acrimonious, debate about the nature of mathematics and its relationship to the rest of the world. The debate took place in articles, published correspondence, and private letters. While it sometimes seemed to generate less light than heat, still, when the smoke cleared, the situation was brighter. The debate led to the careful formulation of various positions, e.g., formalism, Platonism, logicism, and intuitionism, which capture particular aspects of mathematical activity. Few modern mathematicians are terribly happy with any of these positions, but they do seem to have kept the wolf from the door, in that they allowed us to get on with mathematics.

Today a similar controversy about the nature of mathematics and its relation to the rest of the world is raging out of the sight of most mathematicians in the pages of *CACM*, the *Communications of the Association for Computing Machinery*. The debate is almost as exciting and at least as acrimonious. The purpose of this article is to draw the reader's attention to the controversy, and then contribute my two cents worth.

*I would like to acknowledge many conversations with John Etchemendy and Brian Smith over the years on the topic of modeling in mathematics and in computer science. I also need to acknowledge the role that Alan Bush has played in my thinking on the matters discussed in this essay. Talking to him, and watching him at work convinced me that there is something very important about program verification. I would also like to thank various readers of an earlier draft of this article for comments and suggestions.

Background of the Controversy

The present debate swirls around an article called "Program Verification: The Very Idea," [6] written by the philosopher James Fetzer. In this article, the author attacked a certain conception of program verification, and claimed to show that the proclaimed aims of program verification are, in principle, quite simply impossible.

Program verification, theory and practice, is a big business. Millions of dollars are spent on it annually. So rather predictably, there was a large, outraged response. Some of the letters accused Fetzer of misrepresentation, or of knowing nothing at all about program verification. Others accused the editor of publishing an "ill-informed, irresponsible, and dangerous article". On the other hand, some writers found merit in Fetzer's position. One even said that it did not go nearly far enough.* But to understand this debate, and what it has to do with mathematics, we need to back up a few years.

I assume that the reader has an intuitive grasp of notions like algorithm, program, (computing) machine, and implementation. In the way we are using these terms an algorithm $\mathcal{A}$ is an abstract computational process. A program P is a linguistic object that plays a causal role in a computation. In writing a program, one typically tries to implement some (implicit or explicit) algorithm. That is, one wants the program P, when run on a machine M, to carry out the algorithm $\mathcal{A}$. A proof of program correctness would prove that this is the case. (I reserve "computer system correctness" for a stronger notion, which I define towards the end of the article.) Program verification is the business of providing proofs of program correctness.

C. A. R. Hoare is one of the founders of the field of program verification. To see what is at issue, here is a famous quote from Hoare.

> Computer programming is an exact science, in that all the properties of a program and all the consequences of executing it can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. [8]

* These letters can be found in the same issue of *CACM* as [2], [4] and [7].

Since the late 1960s, a great deal of effort has gone into developing program verification techniques.* In spite of two decades of work in the field, Fetzer claims that the premise on which it is based is false.

There was an earlier attack on program verification. In 1979, DeMillo, Lipton and Perlis [3] published a highly critical article. Their contention was that there was a crucial *social* difference between mathematical proofs and proofs of program correctness. In mathematics, proofs are subjected to public scrutiny. There is a social process at work which ends up accepting or rejecting a purported proof. No such process is at work in program verification, they claim. Program verifiers do not publish their purported proofs of correctness and subject them to the test of their peers. They can't. There are just too many programs, the programs are too long, the proofs of correctness too long and boring, and there are too few people interested in reading any such purported proofs. So, they argue, an all important social aspect of mathematical proofs is unavailable in the realm of program verification. The crucial social mechanism for winnowing proofs from faulty purported proofs is unavailable, so the aims of program verification are unsatisfiable.

The DeMillo, Lipton and Perlis article sparked quite a controversy in its own day, as can be imagined, but nothing like the controversy ignited by Fetzer's more recent article. In his article Fetzer did two things. First, he argued that DeMillo, Lipton, and Perlis were mistaken about the nature of proofs, that they conflated genuine formal proofs and "proof sketches". Genuine proofs have certain properties that proof sketches do not have. And with regard to the way mathematics is practiced, he would interpret DeMillo, Lipton and Perlis as saying that most "proofs" published in mathematics papers are simply proof sketches. The social process comes in, Fetzer claimed, in determining which proof sketches are indeed sketches of formal proofs. (We will return to this part of Fetzer's claim later.)

Fetzer's Argument

Fetzer's second aim, and the one which has drawn a firestorm of response, is to show that while DeMillo, Lipton and Perlis were wrong in their reasons, they were correct in their conclusion that program verification is impossible.

Fetzer's argument was summarized in a critical letter (one of the thoughtful reactions) by Bevier, Smith, and Young, roughly as follows:

1. The purpose of program verification is to provide a mathematical method for guaranteeing the performance of a program.

2. This is possible for algorithms, which cannot be executed by a machine, but not possible for programs, which can be executed by a machine.

3. There is little to be gained and much to be lost through fruitless efforts to guarantee the reliability of programs when no guarantees are to be had.

In responding, Fetzer calls their summary of his position perfectly reasonable, "so long as the first premise is intended as a reflection of the position that is - implicitly or explicitly - endorsed by the proponents of program verification."

All three points bear consideration, but we will start with the second. Can one conceivably give a mathematical proof that a given program P, when run on a machine M, will behave properly by being an embodiment of the algorithm $\mathcal{A}$? Or is it, in principle, impossible, as Fetzer claims?

Fetzer's argument for (2) (as he briefly summarized it in [7]) is that computers are

> complex causal systems whose behavior, in principle, can only be known with the uncertainty that attends empirical knowledge as opposed to the certainty that attends specific kinds of mathematical demonstrations. For when the domain of entities that is thereby described consists of purely abstract entities, conclusive absolute verifications are possible; but when the domain of entities that is thereby described consists of nonabstract physical entities ... only inconclusive relative verifications are possible.

Reaction to Fetzer's Argument

Many of the charges leveled against Fetzer's article are typical of encounters between practitioners of any field X and philosophers of X. The philosopher necessarily attempts to give an analysis of X as it presents itself to the informed outsider. The practitioner feels that the philosopher misses a (or the) main point of X. Out of frustration, he is all too often tempted to claim that one simply cannot understand X without doing

* This article will not assume any familiarity with such techniques. However, the reader interested in a simple introduction to them might consult Chapter 4 of [1] or Chapter 5 of [12]. For a more complete look at some of the theoretical issues behind program verification, see [5].

X. As mathematicians (let X be mathematics), we can all surely recognize the temptation. But such reactions do not really tell against the message carried by the philosopher; they simply try to cast doubt or ridicule on the messenger. I will ignore these sorts of reactions to Fetzer's article in what follows and try to get at the substance of the debate, as exemplified by some of the more thoughtful reactions to it.

The argument hinges on a perceived discontinuity between the world of mathematics, and the physical world. It is a gap that Bevier, Smith and Young [2] are at pains to diminish. They argue that if one gives a sufficiently fine-grained analysis of software and hardware, down to the level of logical circuitry, there is little to be made of the distinction. For example, they write ...

> whereas it is true that physical gates do not always behave as their mathematical counterparts ... the semantic gap is sufficiently small to render Fetzer's objections inconsequential. To deny any relation between, say, a physical AND gate and the corresponding Boolean function is to deny that there can be any useful mathematical model of reality. This is tantamount to asserting the impossibility of physical science. [2]

Bevier, Smith and Young also dispute point (1) (and (3), of course). They view program verification not as a branch of mathematics but as a physical science. The goal is not mathematical certainty, but "to make it possible to make highly accurate predictions about the behavior of programs running on computers."

In response to all the mail, the editor of *CACM* invited John Dobson and Brian Randell to try to put the situation into perspective. The result was [4]. Dobson and Randell find blame on both sides of the controversy. They find proponents of program verification guilty of overselling their trade. On the other hand, they accuse Fetzer of failing to observe the distinction between the reasons something *is* so, and the typically weaker reasons one has for *believing it to be* so. In particular, they accuse him of mistaking proofs of program correctness as "providing explanatory reasons for program correctness" whereas they should be taken as providing "merely evidential reasons" for program correctness. They say that "the hypothesis 'this program will execute correctly' is one that can never be proven, only falsified." They take the proof of a program's correctness to show only that certain kinds of errors are not possible.

It bears noting that none of these writers question Fetzer's claim that the aim of program verification is to say something about programs running on physical computers. Their defense is that program verification is a branch of science, not mathematics. So they seem to be backing away from the position staked out by Hoare that got program verification underway. And, in so doing, I think they give up something important too quickly.

Mathematics and the Physical World

At issue is one of the oldest puzzles in the philosophy of mathematics: *How is applied mathematics possible?* On the one hand is the observed fact that mathematics has great efficacy in science as well as in our day-to-day coping with the physical world. On the other hand, there is the seeming divide between the empirical facts of the physical world and the *a priori* nature of the deductive method. Or, from a realist perspective rather than a formalist, there seems to be a divide between the concrete physical objects that populate the physical world and the abstract objects about which we reason in mathematics.

It is this problem around which Fetzer and the more cogent defenders of program verification are warily circling. If there are no mathematical truths about physical objects, then clearly there is a sense in which program verification is impossible, since it would certainly follow that there cannot be a mathematical proof that a given program P, when run on a particular *physical* computer M, has any property whatsoever. On the other hand, applied mathematics is a fact of life. Why shouldn't program verification be a branch of applied mathematics? Whatever makes other forms of applied mathematics possible would surely make program verification possible as well. In what follows, I would like to sketch a picture of what I think is going on in general, and then apply it to three issues in computer correctness.

At the heart of the matter, I think, is the distinction between a given physical (or other) phenomenon and a mathematical model of that phenomenon. It is this distinction which is implicit in Dobson and Randell's "distinction between 'this is the way the world is' and 'this is a useful way of thinking of the world'." It is explicit in Bevier, Smith and Young's claim that Fetzer would have us "deny that there can be any useful mathematical model of reality."

Mathematics, both pure and applied, rests on our experience of the world and our ability to reason. Some of this experience is mathematical experience with various sorts of abstract objects. But ultimately mathematics is grounded in our experience of the nonmathematical world. This claim is controversial, and it is not crucial to the point I want to make, but it does suggest an answer to the question before us. For if mathematics ultimately rests on our experience of the nonmathematical world, it would be odd indeed if mathematics were in principal unable to provide us with any truths about that world.

There are two ways to use mathematics to understand the world. One has become codified in the axiomatic method. We state explicitly various assumptions and then prove that various conclusions follow.* Euclidean geometry is a hackneyed example. There really are things like lines and triangles. Moreover, the axioms of plane geometry are true of them, or close to being true in normal circumstances. Axiomatic set theory is another example. There really are collections of things. Other examples are the axiomatic approaches to the natural or real numbers. In such cases we state as axioms some principles that seem true, or perhaps idealizations of true but messier facts, and prove consequences of the axioms. If our axioms are true, and if the purported proof is correct, then the conclusion must also be true. But what if our axioms are not exactly true? What if they are idealizations? Well, if the world is continuous enough, and if our axioms are close approximations to the facts, then our conclusions will also be close to the facts.

The second and more prevalent method for applying mathematics to the nonmathematical world is that of mathematical modeling. We use some previously established domain of abstract objects, say real numbers, functions, or sets, etc., about which we have an axiomatic theory, to build a model of the new phenomenon under study. For example, the Lebesgue integral is really a mathematical model of a real physical process for determining areas. And most of computer science uses set theory as a tool for modeling computers and other computational structures in the world.

With this form of applied mathematics, there is a somewhat different relationship between theorem and the world. For in this case theorems are really theorems about the domain of mathematical objects used to model the physical world. These truths *can* shed light on the physical world, though. To the extent that our mathematical model is faithful to the phenomenon being modeled, our theorems will correspond to facts about that phenomenon. So again we have at least the possibility of mathematics shedding light on things in the nonmathematical world, for example, determining physical areas or actual computational processes.

Both forms* of applied mathematics have a contingent element (what Fetzer calls "relativity") in their conclusions about the world. The axiomatic method says that our theorems are true *if* our axioms are. The modeling method says that our theorems model facts in the domain modeled *if* there is a close enough fit between the model and the domain modeled. The sad fact of the matter is that there is usually no way to prove – at least in the sense of mathematical proof – the antecedent of a conditional of either of these types.

Still, this does not doom the application of mathematics. After all, our axioms *are* often true, or close enough to the truth. And our mathematical models *are* often good representations of the phenomenon being modeled. Applying mathematics does not in general lend itself to absolute certainty, but it can carry deep and justified conviction. Or, as a philosopher might say, applied mathematics may not guarantee knowledge of facts about the physical world, but it can lead to the next best thing – justified true belief.

How can our conviction in applied mathematics be justified? In answering this, we need to distinguish between what it is that makes a mathematical model (say) fit the world, and what constitutes good evidence for believing that it does. For a model to be a good one there needs to be a mapping from the crucial features of the model to corresponding features of the physical world which is an isomorphism between the model and the modeled, at least in normal circumstances where the model is to be applied.

Sometimes we can have explanatory evidence that there is such a mapping. For example, we have something very close to a mathematical proof that the Lebesgue integral provides a good model of the intuitive notion of area, in normal circumstances. And when modeling something we have built and understand reasonably well, like a particular computer, we may be able to be reasonably certain of the fit between our model and the thing modeled. More typically, though, we only get empirical, experimental evidence that our model is a good one, through the

* This is a great oversimplification of the method, of course. What we really do is to try to systematize fairly self-evident truths about the domain, selecting some to serve as axioms, provided the others follow as consequences.

* Actually, these two forms of applied mathematics are just the ends of a continuum. Often there is some of each going on.

success we have in using it to predict facts in the domain modeled. If experience leads us to trust our model as being a good one, then we will be justified in trusting our theorems as corresponding with the facts of the world. If we are led astray, then we hope that the consequences are not serious, and we revise our model.

Many controversies and errors have arisen out of what one might call *the Fallacy of Identification*, the failure to distinguish between some mathematical model and the thing it is a model of. It is natural enough for the working mathematician to identify the two while he is trying to prove his results, for intuitions about the domain modeled guide us in finding theorems about the mathematical model. But when that final "*QED*" is put at the end of the proof, we must step back and remember that the identification is just that, an identification, not an identity. Fetzer is clear enough about the difference – at least in the case of computers. His antagonists are not so clear.

Fetzer's Argument, Revisited

There is a pervasive ambiguity within the program verification literature. Are the theorems about a mathematical model of computation? Or are they about the real thing? It is a bit hard to say, since the literature is far from clear about the distinction. Workers in program verification sometimes appear to fall prey to the Fallacy of Identification. If pushed to decide between the two, as Fetzer forces them to do, it seems that his antagonists want their work to be about the real thing, not just a model of it, as we noted earlier. They are more willing to give up the mathematical method than they are to give up their claim to be showing something about physically embodied computers. On the other hand, it seems that Hoare's program was meant to apply to mathematical models of computation, and only indirectly, via the model/modeled relation, to the real thing. This view is reinforced by Hoare's more recent principles (see [9]) that "computers are mathematical machines" and "computer programs are mathematical expressions." But whatever Hoare or others *want* to be doing, part of what they *are* doing is proving theorems about their models of computation.

Once things are disambiguated, there seems to be some agreement. Fetzer would agree that mathematical certainty can apply to the mathematical model. The program verification community agrees that mathematical certainty is not possible with regard to the particular physical computers running programs in real time. Where they seem to differ, then, is in how fatal they take the gap between the model and the domain modeled to be. Does the gap lead to Fetzer's pessimistic conclusion (3 above)?

In principle, at least, there is no more or less reason to doubt the applicability of a correct proof of a computer's correctness relative to a mathematical model than to doubt the applicability of any other theorem about some mathematical model of some real world phenomenon, provided we have similar weight of evidence for the appropriateness of the two models. So if there is an argument against the use of mathematical models in program verification, it must involve something problematic about computers, computer programs and computational activity.

The one candidate for a special problem suggested by Fetzer's article stems from the fact that computer programs play a causal role in computation, in that they actually engender the particular process that is carried out. There are two points to be made in response. First, there is the distinction between the program as abstract object, as "type," and the program as part of the physical world, as an instance of the type. It is the latter that plays a causal role. Fetzer is quite explicit in his determination to talk about the latter, but proofs of correctness are about the former. From a syntactic point of view, program instances are simple enough to have very reliable abstract models about which we can prove results. And we can also model the causal role these programs play in computers. At least it is no more problematic than any other form of causality in applied mathematics. In fact, it is rather less problematic, since we know quite well how this takes place, so we can model it quite faithfully. Indeed, as these things go, the models of programs and computers used in proving program correctness are in above average shape. We typically have quite good reasons for trusting them. So if we can come up with a genuine proof that a program is correct, it is typically a pretty reliable indication that the program will indeed compute the intended algorithm.

But this brings us back to a point made by DeMillo, Lipton and Perlis. Real programs are typically very long and complicated. What is the chance of getting a proof of correctness right, especially when there is so little outside interest in checking the proofs?

It seems to me that things are not as bad as they suggest. Admittedly, today's programs are often very long and the proofs can be pretty boring. But there are some telling points on the other side. (1) Program language methodology is advancing to allow programs to have structural features (modularity, typing, etc.)

to aid in their verification. (2) Programs written with eventual verification in mind can make the task simpler. (3) In my (nonempty but admittedly limited) experience, it is rare for such proofs to require great mathematical ingenuity. If they do, then there is something suspicious about the program. And (4), automated proof checking might help in weeding out faulty purported proofs from the real thing. But here a word of caution is in order.

Formal Proofs and Proofs

It seems to me that Fetzer and most of the computer science community, including those involved in automated proof checking, stumble over a landmine left by the retreating formalists. Namely, they fail to recognize the true role of formal languages and formal proofs in mathematics. Fetzer assumes that mathematical logic provides the test of a real proof, with what is usually written in journals being only proof sketches.

> ... a proof of a theorem T, say, occurs just in case theorem T can be shown to be the last member of a sequence of formulae where every member of that sequence is either given ... or else derived from preceding members of that sequence (by relying upon the members of a specified set of rules of inference.)

Here, at the risk of stepping on the toes of my fellow mathematical logicians, I must say that Fetzer, and many others, commit the Fallacy of Identification. That is, they identify a mathematical model of the domain of proofs with that domain itself.

The idea that reasoning could somehow be reduced to syntactic form in a formal, artificially constructed language, is a relatively recent idea in the history of mathematics. It arose from Hilbert's formalist program. There were proofs for thousands of years before logicians came along with the mathematizations of the notion. But these "formal proofs" are themselves certain kinds of mathematical objects: sequences of sentences in a formally specified artificial language, sequences satisfying certain syntactic constraints on their members. They certainly aren't what mathematicians since the time of the ancient Greeks were constructing, for one thing. For another, no particular system can claim to *be* the real notion of proof, since there are endless variations, as is evident from the fact that there are as many different deductive systems as there are textbooks in logic. They can't all be the real notion of proof. Rather, they provide somewhat different models of that notion. And, as Kreisel has observed (**[10]**) in making much the same point, 99% of all mathematicians don't know the rules of even one of these formal systems, but still manage to give correct proofs.

The distinction between formal proofs and real proofs raises the question as to the fit between our mathematical model and the real thing in a new setting. How confident can we be of this fit in this case?

We can be reasonably confident in most of the models, in one sense. We can be reasonably confident that the mathematical objects declared proofs do correspond, under the mapping between the model and the domain modeled, to real proofs, at least if we set aside matters of comprehensibility and elegance. But even here it is worth noting that it took many attempts to get the usual rules of formal logic straight, especially regarding quantifiers and the substitution of terms containing free variables. Many published versions of this model get things wrong, so that objects get counted as proofs (in the model) which do not at all correspond to a valid piece of reasoning.

On the other hand, I think it is clear that current formal models of proof are severely impoverished since there are many perfectly good proofs that are not modeled in any direct way by a formal proof in any current deductive system. For example, consider proofs where one establishes one of several cases and then observes that the others follow by symmetry considerations. This is a perfectly valid (and ubiquitous) form of mathematical reasoning, but I know of no system of formal deduction that admits of such a general rule. They can't, because it is not, in general, something one can determine from local, syntactic features of a proof. Which is just to say that the model of proof accepted by Fetzer (and his opponents) suffers from the very same objection as do models of computers. Which is not to say that it isn't useful. But still, it is a model, not the real thing. And it could be that the best proofs (in the sense of being most enlightening or easiest to understand) of a program's correctness will use methods, like symmetry considerations, that are not adequately modeled in the logician's notion of formal proof, and so which would not be deemed correct by some automated proof checker designed around the formalist's model.

Moreover, identifying proofs with formal proofs leads to what may be an even more serious mistake. Of course to write down a proof that a program P is correct for an algorithm $\mathcal{A}$, we need to have some description, representation, or "specification," of $\mathcal{A}$ itself. The formal model of proofs leads people to suppose that the specification of the algorithm has been given in the artificial language over which the proof regime is defined, usually some descendant of the first-order predicate calculus. While writing things out in complete logical notation can sometimes result in added clarity, all too often it merely obscures things, which is why practicing mathematicians almost never use such a language. And, it is not uncommon for an error to enter the picture in the translation from the English description to the formal specification. Except for the hope of having proofs generated, partially generated, or checked by a computer, there seems to be no compelling reason to specify an algorithm in a formal language. Surely a better practice would be to initially describe the algorithm clearly and unambiguously in the language ordinarily used by mathematicians, an extension of English or some other human language.*

Programs and Computer Systems

As should be clear by now, I disagree with Fetzer's highly pessimistic conclusion (3 above). I think that program verification *is* an effective way of getting more reliable programs. However, there is another problem, if we turn from the question of program verification to the larger question of computer system correctness.

Computer systems are not just physical objects that compute abstract algorithms. They are also embedded in the physical world and they interact with users. They are intended to generate real world activity. The starting assumption of the program verification task is that we are given a mathematical algorithm $\mathcal{A}$, or perhaps some description of it, to implement. But, from the point of view of correctness of the entire system, this begs at least half the question. The larger question of computer system correctness is not whether the machine implements $\mathcal{A}$, but whether the machine carries out the intended real world task. Thus, to solve the larger problem, our mathematical models need to include not just a reliable model of the computer, but also a reliable model of the environment in which it is to be placed, including the user (Smith [**11**]).

Typically, this additional modeling is implicit in the design of the algorithm. But if we are going to divide things up in this way, then a full proof of computer system correctness need consist of not just a proof of program correctness, relative to a reliable model of the computer, but also of a proof of algorithm correctness, relative to a model of the environment in which the system is to be placed. Who really gives a damn that the program correctly computes the algorithm $\mathcal{A}$ if $\mathcal{A}$ is not the one needed in the real computer system.

The question of algorithm correctness is seldom addressed. Partly this stems from the fact that builders of computer systems, like others, find it all too easy to fall into the Fallacy of Identification, forgetting that there can be a gulf between their implicit model and the world. Partly it is because there are fewer known tools at hand to study algorithm correctness. But even if it is addressed, how about the question of the fit between the model and the domain modeled?

Here things are not so rosy. Many "bugs" in programs that make it into general use are not program errors at all. Rather, they result from a failure to anticipate some situations in which the program is required to operate, and some uses the users put it to. These are mismatches between the model of the computer's environment and the computer's actual operating environment.

Modeling the environment and the user is typically *far* more complex than modeling a computer. For example, it often involves many aspects of human psychology, to mention just one nightmare. A good model of the user may need to model her beliefs about the program and how it operates, her likely desires, and her physical abilities. (For example, can she use a mouse quickly and accurately?) Next to modeling this sort of stuff adequately, modeling the computer, or the solar system, is a piece of cake.

In the sort of applications of program verification that really worry Fetzer (like air traffic control or SDI) these problems arise with a vengeance. The unexpected situation is an ever present danger, as is the unusual user. Such cases, being unanticipated by the model builder, often do not fit with the model. There are many documented cases of such mismatches ([**11**]). And if these mismatches take place in a critical computer system we may well not be able to go back and redesign the model. For we may not be around.

In sum, I think Fetzer is correct when he points to the theoretical limitations of program verification. But they're just the limitations implicit in any applied mathematics. Program verification, a branch of applied mathematics, is currently an extremely useful tool for improving the performance of programs, and so of real world computers. On the other hand, proofs of computer system correctness are of value only to the extent that we can rely on the underlying models of

* Another motivation for writing a "formal" specification at some stage might be as a step toward writing the desired program.

the computer and its environment. Hopefully articles critical of computer system correctness methodology, techniques, and standards will not be drowned out, but will help to generate an improved understanding among professionals and the public of what such proofs show and what they do not show about the correctness of physical computers operating in the the real world. For only with such an understanding can those outside the computer science community intelligently assess the relative advantages and dangers of a given proposed use of computer systems.

Bibliography

[1] Bentley, J. *Progamming Pearls*, Addison-Wesley, 1986

[2] Bevier, W.R., Smith, M.K, and Young, W.D. Letter in the Technical Correspondence section of *CACM* 32, 3 (March 1989) 375–376

[3] DeMillo, R., Lipton, R. and Perlis, A. Social processes and proofs of theorems and programs. *CACM* 22, 5 (May 1979) 271–280

[4] Dobson, J. and Randell, B. Program verification: Public image and private reality, *CACM* 32, 4 (April 1989) 420–422

[5] William M. Farmer, Dale M. Johnson, and F. Javier Thayer, "Towards a discipline for developing verified software", in: James H. Burrows and Patrick R. Gallagher, Jr., eds., *Proceedings of the 9th National Computer Security Conference*, September 15–18, 1986 at the National Bureau of Standards, Gaithersburg, Maryland, pp. 91–98; republished in: Rein Turn, ed., *Advances in Computer System Security, Vol. III*, Artech House, Norwood, Massachusetts, 1988, pp. 176–183

[6] Fetzer, James H. Program verification: The very idea, *CACM* 31, 9 (September 1988) 1048–1063

[7] Fetzer, James H. Author's response to various letters about [6] *CACM* 32, 3 (March 1989) 377–381

[8] Hoare, C.A.R. An axiomatic basis for computer programming. *CACM* 12 (1969), 576–580

[9] Hoare, C.A.R. Mathematics of programming, BYTE (August 1986), 112–149

[10] Kreisel, G. Informal rigour and completeness proofs, Int. Colloq. Philos. Sci., 1 (1965) 138–186

[11] Smith, B.C. The Limits of Correctness, SIGCAS Newsletter 14, 4 (December 1985) 18–26

[12] Wulf, W.A., Shaw, M., Hilfinger, P.N. and Flon, L. *Fundamental Structures of Computer Science*, Addison-Wesley, 1981

Computers and Mathematics

Edited by Jon Barwise

Editorial notes

I start off this month by admitting two embarrassing lapses.

Who developed those programs?

I have a set of guidelines I send out to software reviewers. One thing I forgot to mention in those guidelines was that a review should always indicate who developed the software. A couple of recent reviews have failed to do this, and readers have let me know about it in no uncertain terms. I regret these omissions and have added a new sentence to my guidelines.

One omission, ironically, was in the review of *Tarski's World* by Mark Seligman in the November 1989 issue. It happens that this logic courseware program was developed by John Etchemendy and me, with the support of the FAD program at Stanford University.

The other was in the review of *Exploring Small Groups* by Suzanne Molnar in the December 1989 issue. This program was developed by Ladnor Geissinger, who also wrote the manual. Geissinger is Professor of Mathematics at the University of North Carolina at Chapel Hill and a Fellow of the Institute for Academic Technology. He was given programming and technical support by an IBM/UNC software development project grant.

Correspondence

More on proving computer correctness

In the September 1989 column I wrote a piece reporting on a debate over proofs of program correctness. I also attempted to shed some light on that debate by appealing to the distinction between real world phenomena, and mathematical models of them. I analyzed the problem as a failure to distinguish carefully between the two.

By and large, the response to this article has been quite positive. A number of people in the program correctness community have said that it somehow managed to both shed light on, and cool, the controversy. By contrast, Richard Dudley of the M.I.T. mathematics department writes:

Program Verification

Barwise [1] reports on a discussion among philosophers, especially Fetzer [4], computer scientists, and now mathematicians: can correctness of executable computer programs be proved? But essentially everyone agrees that it cannot be proved, in the strict mathematical sense, that a physical computer will execute a program correctly. A problem has been raised by use of the word "proof" about program verification. There is wide agreement that program verification is (currently) part of applied rather than pure mathematics, but mathematicians may well think of proofs as characteristic of pure mathematics. One might, for example, predict the orbit of a satellite very accurately without claiming to *prove* that it will be in a specific small region at a future time. I suppose very few applied mathematicians or scientists would claim to prove, in the sense of mathematical proof, anything about the physical world.

If, as Dobson and Randell [3] well say, "the hypothesis 'this program will execute correctly' is one that can never be proven, only falsified", and a 'proof' of a program's correctness shows "only that certain kinds of errors are not possible", then we are dealing with relatively weak, perhaps new (to mathematicians) meanings of "proof" and/or "correctness," which one should be clear about. Other central mathematical notions such as equality have acquired new meanings in computing: the computer-language equation $S = S + x$, where x is not 0, for example, in the context of a summation loop, might be understood as the mathematical equation $S(n) = S(n-1) + x(n)$, while in many computer languages "$S + x = S$" is a syntax error.

On another point, actual programs are usually written in higher-level languages such as Fortran or C, then translated by a compiler or interpreter into a lower-level language and executed in connection with operating system software. Such systems programs may in turn have been written in higher-level languages and compiled by (another or partial) compiler. Systems software (and hardware) provide the environment about which Hoare [6] wrote: "Computer programming is an exact science in that all the properties of a program and all the consequences of executing it *in any given environment* can, in principle, be found out from the text of the program itself by means of purely deductive reasoning" [emphasis added]. Unfortunately Barwise [1] calls this a "famous quotation" but omits the phrase I emphasized, without even an ellipsis (...). Even if the misquote was found in another secondary source, the role of the environment should not have been overlooked. Fetzer [5] also mentions and criticizes the "famous passage" without noting the misquotation.

Fetzer [4] was, in turn, primarily a reaction to DeMillo, Lipton and Perlis [2], who said more or less that correctness of programs was not being proved effectively in the 1970s because people were not checking others' proofs. Fetzer thought that one should go further and say that proofs of program correctness are not possible. For opinions in favor of such proofs both Fetzer [4,5] and Barwise [1] cite Hoare [6,7]. Fetzer [4] quoted selectively from Hoare [7], who did write:

"I hold the opinion that the construction of computer programs is a mathematical activity like the solution of differential equations, that programs can be derived from their specifications through mathematical insight, calculation, and proof, using algebraic laws as simple and elegant as those of elementary arithmetic... . Computers are mathematical machines... computer programs are mathematical expressions ... a programming language is a mathematical theory ... programming is a mathematical activity."

But neither Fetzer nor Barwise tells us that Hoare [7] went on as follows:

"HOWEVER ... [emphasis in original]

These are general philosophical and moral principles, but all the actual evidence is against them. Nothing is as I have described it, neither computers nor programs nor programming languages nor even programmers."

I'm afraid Barwise and Fetzer have done us a disservice by their incomplete quotations. But at the beginning of [4], Fetzer wrote "There are those, such as Hoare ... who maintain that computer programming should strive to become more like mathematics." That, I believe, is a fair summary of what Hoare was actually saying, and it may be arguable, but in full and in context I think Hoare was addressing a question of what will work best in the future for computer programmers. Hoare [7], even according to one quote given in Fetzer [4, p. 1058] (but not Fetzer [5] or Barwise [1]) was negative about proofs of program correctness in typical current environments.

There are mathematically interesting and difficult issues in precisely deriving a program from its specifications. Even if the specification calls for evaluating a given polynomial, the results are non-unique since in current computer arithmetic addition, done to a fixed number of binary or decimal places, is not associative. It is unfortunate that these real issues were obscured in the philosophical discussion.

References

1. Barwise, J. Mathematical proofs of computer system correctness. *Notices* **36** (1989), 844–851.

2. DeMillo, R., Lipton, R., and Perlis, A. Social processes and proofs of theorems and programs. CACM **22** #5 (May 1979), 271–280.

3. Dobson, J. and Randell, B. Program verification: Public image and private reality. CACM **32**, 4 (April 1989) 420–422.

4. Fetzer, J. H. Program verification: the very idea. CACM **31** (1988) 1048–1063.

5. Fetzer, J. H. (letter). *Notices* **36** (1989) 1352-1353.

6. Hoare, C.A.R. An axiomatic basis for computer programming. CACM **12** (1969), 576–580, 583.

7. Hoare, C.A.R. Mathematics of programming. *Byte*, August 1986, 115–121.

Reply: I plead guilty to misquoting Hoare, omitting the phrase "in any given environment." I simply took the quote from Fetzer's article without checking the original. Fetzer tells me that there was an elipsis in earlier versions of his article, but that it somehow disappeared along the way. If the missing words are replaced, the ambiguity between the real world and the mathematical phenomena persists, since the term "environment" has two meanings. One reading would take it to be the actual environment in which a program is run on a physical computer. The other usage is where environments are certain abstract mathematical objects. Both are quite common in computer science. In terms of my analysis of the larger debate, the term "environment" is itself ambiguous between the physical environment, and a mathematical model of it, or rather, of certain aspect of it omitted from the model of the computer itself. If we interpret all this in terms of the real thing, then Fetzer's argument applies and proving programs correct is impossible. If we interpret it as applying to the mathematical model, then it is possible, but only as useful as the fit between the model and the real thing. Which Hoare had in mind, if he was in fact clear about the distinction, does not seem to important. For the point of my piece was not to attack or defend Hoare or Fetzer or anyone else, but to try to illuminate a controversial special case of applied mathematics.

Uses of computers in mathematics

This portion of the column is devoted to short articles detailing ways mathematicians have found to use computers in some aspect of mathematics: teaching, research, writing, ... Readers are invited to submit articles to the editor: Jon Barwise, CSLI, Ventura Hall, Stanford, CA 94305, or in LaTeX by email at: barwise@csli.stanford.edu.

Computers in Mathematics at Lafayette College

Clifford A. Reiter and Thomas R. Yuster
Lafayette College

The computational environment at Lafayette is different than at Grinnel and the University of Wisconsin-Madison as described in this column by Gene Herman in March 1989 and Rod Smart in May/June 1989. Yet there are some obvious similarities in the hardware and instructional use. You will see that our department is active in using computing in teaching but does not have any grand programs (yet). We have acquired most of our equipment with support from the college administration and Pennsylvania state grants. The department has been active in letting the administration know its needs.

Lafayette College has just under 2000 full time undergraduate students and a small part time program but no graduate program. Engineering accounts for 20–30% of the student body. Computer Science is a separate department. The mathematics department has 16 full time faculty. About ten sections of the scientific calculus sequence are taught each semester with 24 students per