PROGRAM VERIFICATION: THE VERY IDEA

The notion of program verification appears to trade upon an equivocation. Algorithms, as logical structures, are appropriate subjects for deductive verification. Programs, as causal models of those structures, are not. The success of program verification as a generally applicable and completely reliable method for guaranteeing program performance is not even a theoretical possibility.

JAMES H. FETZER

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.

C. A. R. Hoare

There are those, such as Hoare [20], who maintain that computer programming should strive to become more like mathematics. Others, such as DeMillo, Lipton and Perlis [8], contend this suggestion is mistaken because it rests upon a misconception. Their position emphasizes the crucial role of social processes in coming to accept the validity of a proof or the truth of a theorem, no matter whether within purely mathematical contexts or without: "We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem" [8, p. 271]. As they perceive it, the situation with respect to program verification is worse insofar as no similar social process occurs between program verifiers. The use of verification to guarantee the performance of a program is therefore bound to fail. Although Hoare's work receives scant attention in their paper, there should be no doubt that his approach—and that of others, such as E. W. Dijkstra [10], who share a similar point of view—is the intended object of their criticism.

Their presentation has aroused enormous interest and considerable controversy, ranging from unqualified agreement [expressed, for example, by Glazer [13]: "Such an article makes me delight in being . . . a member of the human race"] to unqualified disagreement [expressed, for example, by Maurer [28]: "The catalog of criticisms of the idea of proving a program correct . . . deserves a catalog of responses . . ."]. Indeed, some of the most interesting reactions have come from those whose position lies somewhere in between, such as van den Bos [37], who maintains that, "Once one accepts the quasi-empiricism in mathematics, and by analogy in computer science, one can either become an adherent of the Popperian school of conjectures (theories) and refutations [32], or one may believe Kuhn [23], who claims that the fate of scientific theories is decided by a social forum . . .".[1] Perhaps better than any other commentator, van den Bos seems to have put his finger on what may well be the crucial issue raised by [8], namely: if program verification, like mathematical validation, could only occur as the result of a fallible social process, if it could occur at all, then what would distinguish programming procedures from other expert activities, such as judges deciding cases at law and referees reviewing articles for journals? If it is

[1] Popper advocates the conception of science as an objective process of "trial and error" whose results are always fallible, while Kuhn emphasizes the social dimension of scientific communities in accepting and rejecting what he calls "paradigms." See, for example, [23], [32–33]. A fascinating collection of papers discussing their similarities and differences is presented in [25].

naive to presume that mathematical demonstrations, program verifications and the like are fundamentally distinct from these activities, on what basis can they be differentiated?

The purpose of this article is to investigate the arguments that DeMillo, Lipton and Perlis have presented in an effort to disentangle several issues that seem to have become intricately intertwined. In particular, their position, in part, rests upon a difference in social practice that could change if program verifiers were to modify their behavior. It also depends, in part, upon problems that arise from the complexity of the programs to be verified. There appear to be two quite different kinds of "program complexity," however, only one of which succumbs to their arguments. Moreover, if program verifiers were to commence collaborating in their endeavors, the principal rationale underlying their position would tend to disappear. Indeed, while social processes are crucial in determining what theorems the mathematical community takes to be true and what proofs it takes to be valid, they do not thereby make them true or valid. The absence of similar social processes in determining which programs are correct, accordingly, does not affect which programs are correct. Nevertheless, there are reasons for doubting whether program verification can succeed as a generally applicable and completely reliable method for guaranteeing the performance of a program. Therefore, it looks as though DeMillo, Lipton and Perlis have offered some bad arguments for some positions that need further elaboration and deserve better support.

MATHEMATICS AS A FALLIBLE SOCIAL PROCESS

> *Outsiders see mathematics as a cold, formal, logical, mechanical, monolithic process of sheer intellection; we argue that insofar as it is successful, mathematics is a social, informal, intuitive, organic, human process, a community project.*
>
> DeMillo, Lipton and Perlis

The conception of mathematical procedure portrayed by DeMillo, Lipton and Perlis initially drew a distinction between *proofs* and *demonstrations*, where demonstrations are supposed to be long chains of formal logic, while proofs are not. The difference intended here, strictly speaking, appears to be between proofs and what are typically referred to as "proof sketches," where proof sketches are incomplete (or "partial") proofs. Indeed, proofs are normally defined in terms of demonstrations, where a proof of 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 (as an axiom or as an assumption) or else derived from preceding members of that sequence (by relying upon the members of a specified set of rules of inference) [6, p. 182]. In fact, what is known as *mathematical induction* is a special case of the application of demonstrative procedures to infinite sequences, where these processes, which tend to rely upon recursive techniques, are completely deductive [2, p. 169].

Moreover, when DeMillo, Lipton and Perlis offer "proof sketches" as the objects of mathematicians' attention rather than proofs, it becomes possible to make (good) sense of otherwise puzzling statements such as:

> In mathematics, the aim is to increase one's confidence in the correctness of a theorem, and it's true that one of the devices mathematicians could in theory use to achieve this goal is a long chain of formal logic. But in fact they don't. What they use is a proof, a very different animal. Nor does the proof settle the matter; contrary to what its name suggests, a proof is only one step in the direction of confidence. [8, p. 271]

Thus, while a proof, strictly speaking, is a (not necessarily long) chain of formal logic that is no different than a demonstration, a proof sketch is "a very different animal," where a proof sketch, unlike a proof, may often be "only one step in the direction of confidence." Nevertheless, although these reflections offer an interpretation under which their statements appear to be true, it leaves open a larger question, namely: whether the aim of proofs in mathematics can be adequately characterized as that of "increasing one's confidence in the correctness of theorems" rather than as formal demonstrations.

DeMillo, Lipton and Perlis have offered some bad arguments for some positions that need further elaboration and deserve better support.

In support of their depiction, DeMillo, Lipton and Perlis emphasize the tentative and fallible character of mathematical progress, where out of some 200,000 theorems said to be published each year, "A number of these are subsequently contradicted or otherwise disallowed, others are thrown into doubt and most are ignored" [8, p. 272]. Since numerous purported proofs are unable to withstand critical scrutiny, they suggest, the acceptability or believability of a specific mathematical result depends upon its reception and ultimate evaluation by the mathematical community. In this spirit, they describe what appears to be a typical sequence of activity within this arena, where, say, a proof begins as an idea in someone's mind, receives translation into a sketch, is discussed with colleagues and, if no substantial objections arise, is developed and submitted for publication, where, if it survives the criticism of other mathematicians, then it tends to be accepted [8, p. 273].

In this sense, the behavior of typical members of the mathematical community in the discovery and promotion of specific findings certainly assumes the dimensions of a social process involving more than one person interacting together to bring about a certain outcome. Although it may be difficult to imagine a Bertrand Russell or a David Hilbert rushing to his colleages for their approval of his findings, there would appear to be no good reasons to doubt that average mathematicians frequently behave in the manner described. Therefore, I would tend to agree that mathematicians' mistakes are typically discovered or corrected through casual interaction with other mathematicians. The restraints imposed by symbolic logic, after all, exert their influence only through their assimilation as habits of thought and as patterns of reasoning by specific members of a community of this kind: discoveries and corrections of mistakes usually occur when one mathematician gently nudges another "in the right direction." (Relevant discussions can be found in [4] and [24].)

To the extent to which DeMillo, Lipton and Perlis should be regarded as endorsing the view that review procedures exercised by colleagues and peers tend to improve the quality of papers that appear in mathematics journals, there seems to be little grounds for disagreement. For potential proofs are often strengthened, theorems altered to correspond to what is provable, and various arguments discovered to be deeply flawed through social interaction. Nevertheless, a community of mathematicians who are fast and sloppy referees is not especially difficult to imagine: where is the university whose faculty do not occasionally compose shoddy and inaccurate reviews—even for very good journals? After all, what makes (what we call) a *proof* a proof is its validity rather than its acceptance (by us) as valid, just as what makes a sentence true is what it asserts to be the case is the case, nor merely that it is believed (by us) and therefore referred to as *true*. Social processing, therefore, is neither necessary nor sufficient for a proof to be valid, as DeMillo, Lipton and Perlis implicitly concede [8, p. 272].

DEDUCTIVE VALIDITY AND PSYCHOLOGICAL CERTAINTY

> *. . . a theorem either can or cannot be derived from a set of axioms. I don't believe that the correctness of a theorem is to be decided by a general election.*
>
> L. Lamport

Confidence in the truth of a theorem (or in the validity of an argument), of course, appears to be a psychological property of a person-at-a-time: one and the same person at two different times can vary greatly in his confidence over the truth of the same theorem or the validity of the same argument, just as two different persons at the same time might vary greatly in their confidence that that same theorem is true or that that same argument is valid. Indeed, there is nothing inconsistent about scenarios in which, say, someone is completely confident that a specific formula is a theorem (when it happens to be false) or else completely uncertain whether a particular argument is valid (when it happens to be valid). No doubt, mathematicians are sometimes driven to discover demonstrations of theorems after they are already completely convinced of their truth. Demonstrations, in such cases, cannot increase the degree of confidence when that degree is already maximally strong. But that is not to deny they can still fulfill other—non-psychological—functions, such as providing objective evidence of the truth of one's subjective belief.

From the point of view of a traditional theory of knowledge, the role of demonstration becomes readily apparent; for the classical conception of knowledge characterizes "knowledge" in terms of three necessary and sufficient conditions as warranted, true belief (for example, [7, ch. 2]). Hence, an individual z who is in a state of belief with respect to a certain formula f, where z believes that f is a theorem, say, cannot be properly qualified as possessing knowledge that f is a theorem unless his belief can be supported by means of reasons, evidence, or warrants, which might be one or another of three different kinds, depending upon the nature of the objects that might be known. For results in logic and mathematics fall within the domain of deductive methodology and require demonstrations. Lawful and causal claims fall within the domain of empirical inquiries and require inductive warrants. Observational and experimental findings fall within the domain of perceptual investigations and acquire support on the basis of direct sense experience.[2]

With respect to deductions, the term *verification* can be used in two rather different senses. One of these occurs in pure mathematics and in pure logic, in which theorems of mathematics and of logic are subject to demonstration. These theorems characterize claims that are always true as a function of the meanings assigned to the specific symbols by means of which they are expressed. Theorem-schemata and theorems in this sense are subject to verification by deriving them from no premises at all (within systems of natural deduction) or from primitive axioms (within axiomatic formal systems).[3] The other occurs in ordinary reasoning and in scientific contexts in general, whenever *conclusions* are shown to follow from specific sets of *premises*, where there is no presumption that these conclusions might be derived from no premises at all or that those premises should be true as a function of their meaning. Thus, within a system of natural deduction or an axiomatic formal system, the members of the class of consequences that can be derived from no premises

[2] The point is that there appear to be three kinds of evidence that can be advanced in support of the truth of a sentence, two of which occur in the form of other sentences (whose truth may require their own establishment). While descriptions of the data of more-or-less direct experience presupposes causal interactions between language-users and the world, the construction of arguments does not. [11 esp. pp. 18–24]

[3] For an exceptionally lucid discussion of the differences between so-called natural deduction systems and axiomatic formal systems, where the former operate with sets of inference rules in lieu of formal axioms but the latter operates with sets of axioms and as few as a single rule, see [3].

at all or that follow from primitive axioms alone may be said to be *absolutely* verifiable. By contrast those members of the class of consequences that can only be derived relative to specific sets of premises whose truth is not absolutely verifiable, may be said to be *relatively* verifiable.[4]

The difference between absolute and relative verifiability, moreover, is extremely important for the theory of knowledge. For theorems that can be verified in the absolute sense cannot be false, so long as the rules are not changed or the axioms are not altered. But conclusions that are verified in the relative sense can still be false (even when the premises and the rules remain the same). The absolute verification of a theorem thus satisfies both necessary and sufficient conditions for its warranted acceptance as true, but the relative verification of a conclusion does not. Indeed, as an epistemic policy, the degree of confidence that anyone should invest in the conclusion of an argument should never exceed the degree of confidence that ought to be invested in its premises—even when it is valid! Unless the premises of an argument cannot be false—unless those premises themselves are absolutely verifiable—it is a mistake to assume the conclusion of a valid argument cannot be false. No more can appropriately be claimed than that its conclusion must be true if all its premises are true, which is the defining property of a valid demonstration.

CONSTRUCTING PROOFS AND VERIFYING PROGRAMS

> *Formal proofs carry with them a certain objectivity. That a proof is formalizable, that the formal proofs have the structural properties that they do, explains in part why proofs are convincing to mathematicians.*
>
> T. Tymoczko

The truth of the conclusion of a valid deductive argument, therefore, can never be more certain than the truth of its premises—unless its truth can be established on other grounds. While deductive reasoning preserves the truth (insofar as the conclusion of a valid argument cannot be false if its premises are true), the truth of those premises can be guaranteed, in general, only under those special circumstances that arise when they themselves are verifiable in the absolute sense. Otherwise, the truth of the premises of any argument has to be established on independent grounds, which might be deductive, inductive or perceptual. Yet none of these types of warrants provides an infallible foundation for any inference to the truth of the conclusions they support. Perhaps few of us would be inclined to think that our senses are infallible, i.e., that things must always be the way they appear to be. The occurrence of illusions, hallucinations and delusions abuses us of that particular fantasy. The mistakes we make about reasoning are far more likely to occur concerning inductive and deductive arguments, whose features are frequently not clearly understood. We should not overlook that, apart from imagination and conjecture, which serve as sources of ideas but do not establish their truth, all of our states of knowledge—other than those of pure mathematics and logic—are ultimately dependent for their support upon direct and indirect connections to experience.

The difference between absolute and relative verifiability is extremely important for the theory of knowledge.

The features that distinguish (good) deductive arguments are the following:

(a) they are *demonstrative*, i.e., if their premises were true, their conclusions could not be false (without contradiction);
(b) they are *non-ampliative*, i.e., there is no information or content in their conclusions that is not already contained in their premises; and,
(c) they are *additive*, i.e., the addition of further information in the form of additional premises can neither strengthen nor weaken these arguments, which are already maximally strong.

Thus, the non-ampliative property of (good) deductive arguments can serve to explain both their demonstrative and additive characteristics. Demonstrative arguments, of course, are said to be "valid," while valid arguments with true premises are said to be "sound" (and cannot possibly have false conclusions).[5]

Compared to deductive arguments, (good) inductive arguments are (a) *non-demonstrative*, (b) *ampliative*, and (c) *non-additive*. Arguments satisfying appropriate inductive standards likewise should be said to be "proper," while proper arguments with true premises may be said to be "correct" (but can have conclusions that are false even when their premises are true). It should come as no surprise, therefore, that the purposes served by such different types of arguments are quite distinct, indeed. For inductive arguments are meant to be *knowledge-expanding*, while deductive arguments are meant to be *truth-preserving*. The ways in which inductive arguments expand our knowledge assume various forms. Reasoning from samples to populations, from the observed to the unobserved and from the past to the future always involves drawing inferences to conclusions that contain more information or content than do

[4] The claim that a proposition has been "verified," therefore, can mean more than one thing, since some derivations establish that a claim is a theorem (which cannot possibly be false), while other derivations establish that a claim is a conclusion (which cannot be false, so long as its premises are true). Derivations of both types are equally "valid," of course, but only the former show that their conclusions cannot be false.

[5] This point differs from the previous note. That the valid conclusion of an argument from true premises cannot be false does not mean that we can know, in general, whether or not a valid argument does or does not have true premises. Thus, the difficulty we confront, from an epistemic point of view, is establishing the truth of our premises as well as the validity of our arguments. The importance of this difference is great.

their premises. The most familiar instances of inductive reasoning we all employ in our daily lives concern the behavior of ordinary physical things—things that may or may not work right, fit properly, or function smoothly (such as electrical appliances, including microwave ovens and personal computers). When we interact with systems such as these, we invariably base our expectations upon our experience: we draw conclusions concerning their behavior in the future from their behavior in the past. All such reasoning is ampliative and—as we all too often discover to our dismay—is both non-demonstrative and non-additive.

The function of a program, of course, is to convey instructions to a computer. Most programs today are written in high-level programming languages, such as Pascal, LISP and Prolog, which simulate "abstract machines," whose instructions can be more readily composed than can those of the machines that ultimately execute them. Thus, a source program written in a high-level language is translated into an equivalent low-level "object program" written in machine language either directly, by an interpreter; or indirectly, by a compiler. The "target machine" then executes the object program when instructed to do so. If programs are verifiable, therefore, then they must be subject to deductive procedures. Indeed, precisely this conception is advanced by [19, p. 576]:

> 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.

Thus, if programs are absolutely verifiable, then there must exist some *program rules of inference* or *primitive program axioms* permitting inferences to be drawn concerning the performance that a machine will display when such a program is executed. If they are relatively verifiable, then there must be sets of premises concerning the text of such a program from which it is possible to derive conclusions concerning the performance that that machine will display when that program is executed. If these conditions cannot be satisfied, however, then the very idea of program verification will have been misconceived.

SOCIAL PROCESSES AND PROGRAM VERIFICATIONS

> *We do not argue that strict logical deduction should be the only way that mathematics should be done, or even that it should come first; rather, it should come last, after the theorems to be proved, and their proofs, are well understood.*
>
> W. D. Maurer

The critical dimension of deduction and induction, furthermore, is the justification of corresponding classes of rules as acceptable principles of inference. For, in their absence, it would be impossible to ascertain which, among all those arguments that—rightly or wrongly—are supposed to be valid or proper, actually are. Within this domain, "thinking doesn't make it so." The fact that a community of mathematicians happens to agree upon the validity of an argument or the truth of a theorem, alas, no more guarantees the validity of that argument or the truth of that theorem than agreement within a society of observers that the Earth is flat (for which a variety of mutually convincing arguments are advanced) could guarantee that that belief is true. In the absence of classes of rules of inference whose acceptability can be justified, in other words, validity and propriety are merely subjective properties of arguments insofar as specific persons happen to hold them in high esteem, where their standing may vary from person to person and from time to time.

Since computer programs, like mathematical proofs, are syntactical entities consisting of sequences of lines (strings of signs and the like), they both appear to be completely formalized entities for which completely formal procedures appear to be appropriate. Yet programs differ from theorems, at least to the extent to which programs are supposed to possess a semantic significance that theorems seem to lack. For the sequences of lines that compose a program are intended to stand for operations and procedures that can be performed by a machine, whereas the sequences of lines that constitute a proof do not.[6] Even if the social acceptability of a mathematical proof is neither necessary nor sufficient for its validity, the suggestion might be made that the existence of social processes of program verification may actually be even more important for the success of this endeavor than it is for validating proofs. The reason that could be advanced in support of this position is the opportunity that such practices would provide for more than one programmer to inspect a program for a suitable relationship between its syntax and its intended semantics, i.e., the behavior expected of the machine.

There are no reasons for believing that DeMillo, Lipton and Perlis are mistaken in their observation that the verification of programs is not a popular pastime [8]. The argument we are considering thus serves to reinforce the importance of this disparity in behavior between the members of the mathematical community and the members of the programming fraternity. Their position implicitly presumes that programmers are inherently unlikely to collaborate on the verification of programs, if only because it is such a tedious and complex activity. Suppose, however, that conditions within society were to change in certain direct and obvious ways, say that substantial financial rewards were offered for the best team efforts in verifying programs, with prizes of up to $10,000,000 awarded by Ed McMahon and guest appearances on *The Tonight*

[6] It could be argued, of course, that even these purely syntactical entities may be viewed as having semantical relations to some domain of abstract entities. In the case of pure mathematics, these entities might be variously entertained as Platonic, as Intuitive, or as Conventional. See, for example, the discussions found in [1] The key issue, however, is not whether there might be some such abstract domain of interpretation but whether there is any such physical domain.

Show. Surely under circumstances of this kind, the past tendency of program verifiers to engage in solitary enterprise might be readily overcome, resulting in the emergence of a new wave in program verification, "the collaborative verification group," dedicated to mutual efforts by more than one programmer to verify particular programs. Surely under these conditions—which are not completely far-fetched in the context of modern times—a social process for the verification of programs could emerge within the computer science community that would be the counterpart to the social process for the validation of proofs in mathematics. Under these circumstances, there would not be any difference of this kind.

If this were to come about, then the primary assumption underlying the position of DeMillo, Lipton and Perlis would no longer apply. Regardless of what other differences might distinguish them, in this respect their social processing of theorems and of programs would be the same. If we refer to differences between subjects or activities that could not be overcome, no matter what efforts we might undertake, as differences "in principle," and to those that could be overcome, by making appropriate efforts, as differences "in practice," then it should be obvious that DeMillo, Lipton and Perlis have identified a difference in practice that is not also a difference in principle. What this means is that to the extent to which their position depends upon this difference, it represents no more than a contingent, *de facto* state-of-affairs that might be a mere transient stage in the development of program verification within the computer science community. If there is an "in principle" difference between them, it must lie elsewhere, because divergence in social practice is a difference that could be overcome.

THE CONCEPTION OF PROBABILISTIC PROOFS

> *If proofs bear little resemblance to formal deductive reasoning, if they can stand for generations and then fall, if they can contain flaws that defy detection, if they can express only the probability of truth within certain error bounds—if they are, in fact, not able to prove theorems in the sense of guaranteeing them beyond probability and, if necessary, beyond insight, well, then, how does mathematics work?*
>
> DeMillo, Lipton and Perlis

When DeMillo, Lipton and Perlis [8, p. 273] advance the position that "a proof can, at best, only probably express truth," therefore, it is important to discover exactly what they mean, since proofs are deductive and accordingly enjoy the virtues of demonstrations. There are several alternatives. In the first place, this claim might reflect the differences that obtain between proofs and proof sketches, insofar as incomplete or partial proofs do not satisfy the same objective standards—and therefore need not convey the same subjective certainty—as do complete proofs. In the second place, it might reflect the fallibility of acceptance of the premises of such an argument—which could be valid yet have at least one false premise and therefore be unsound—because of which acceptance of its conclusion should be tempered with uncertainty as well. While both of these ideas find expression in their article, however, their principal contention appears to be of a rather different character altogether.

Thus, DeMillo, Lipton and Perlis distinguish between the "classicists" and the "probabilists," where classicists maintain that:

> . . . when one believes mathematical statement *A*, one believes that in principle there is a correct, formal, valid, step by step, syntactically checkable deduction leading to *A* in a suitable logical calculus . . .

which is a complete proof lying behind a proof sketch as the object of acceptance or of belief. Probabilists, by comparison, instead maintain that:

> . . . since any very long proof can at best be viewed as only probably correct, why not state theorems probabilistically and give probabilistic proofs? The probabilistic proof may have the dual advantage of being technically easier than the classical, bivalent one, and may allow mathematicians to isolate the critical ideas that give rise to uncertainty in traditional, binary proofs.

Thus, in application to proofs or to proof sketches, there appear to be three elements to this position: first, that long proofs are difficult to follow; second, that probabilistic proofs are easier to follow; and, third, that probabilistic proofs may disclose the problematic aspects of ordinary proofs. Rabin's algorithm for testing probable prime numbers is offered as an illustration, where, "if you are willing to settle for a very good probability that *N* is prime (or not prime), then you can get it within a reasonable amount of time—and with (a) vanishingly small probability of error."

Their reference to "traditional, binary proofs" is important insofar as traditional proofs (in the classical sense) are supposed to be either valid or invalid: there is nothing "probable" about them. Therefore, I take DeMillo, Lipton and Perlis to be endorsing an alternative conception, according to which arguments are amenable to various measures of strength (or corresponding "degrees of conviction"), which might be represented by, say, some number between zero and one inclusively, with some of the properties associated with probabilities, likelihoods, or whatever [11, Part III]. A hypothetical scale could be constructed accordingly, such that degrees of partial support of measures zero and one for instance are distinguished from worthless fallacies (whose premises, for example, might be completely irrelevant to their conclusions), on the one hand, and from demonstrative arguments (the truth of whose premises guarantees the truth of their conclusions), on the other, as extreme cases not representing *partial* support.

From this perspective, the existence of even a "vanishingly small" probability of error is essential to a probabilistic proof. If there were no probability of error at all a proof could not be probabilistic. Thus, if the commission of an error were either a necessity (as in the case of a fallacy of irrelevance) or an impossibility (as in the case of a valid demonstration), then an argument would have to be other than probabilistic. Indeed, the existence of fallacies of irrelevance exemplifies an important point discussed above: fallacious arguments, though logically flawed, can nevertheless exert enormous persuasive appeal—otherwise, we would not have to learn how to detect and avoid them [29, Ch 10]. But, if this is the case, then DeMillo, Lipton and Perlis, when appropriately understood, are advocating a conception of mathematics according to which (a) classical proofs are practically impossible (so that demonstrations are not ordinarily available), yet (b) worthless fallacies are still unacceptable (so that our conclusions are nevertheless supported). This position strongly suggests the possibility that DeMillo, Lipton and Perlis should be viewed as advocating the conception of mathematics as a domain of inductive procedure.

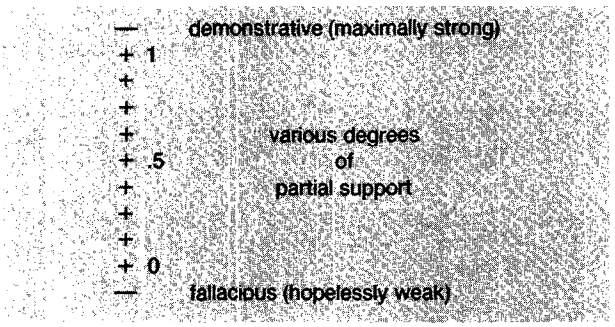

FIGURE 1. A Measure of Evidential Support

When consideration is given to the distinguishing characteristics of inductive arguments, this interpretation seems to fit their position quite well. As stated earlier, inductive arguments are (a) nondemonstrative, (b) ampliative, and (c) nonadditive. This means (a) that their conclusions can be false even when their premises are true (permitting the possibility of error); (b) that their conclusions contain some information or content not already contained in their premises (otherwise they would not be nondemonstrative); and, (c) the addition of further evidence in the form of additional premises can either strengthen or weaken these arguments (whether that evidence is discovered days, months, years or centuries later). Thus, in accepting the primality of some very large number N probabilistically, for example, one goes beyond the content contained in the premises (which do not guarantee the truth of that conclusion) and runs a risk of error (which cannot be avoided with probabilistic reasoning). Yet one thereby possesses evidence in support of the truth of such a conclusion, so that its acceptance is warranted to some degree. After all, that might be the best we can do.

MATHEMATICAL PROOFS AND PROBABLE VERIFICATIONS

Mathematical proofs increase our confidence in the truth of mathematical statements only after they have been subjected to the social mechanisms of the mathematical community. These same mechanisms doom the so-called proofs of software, the long formal verifications that correspond, not to the working mathematical proof, but to the imaginary logical structure that the mathematician conjures up to describe his feeling of belief.

DeMillo, Lipton and Perlis

Indeed, whether or not we can do better appears to be at the heart of the controversy surrounding this position. DeMillo, Lipton and Perlis, after all, do not explicitly deny the existence of classical bivalent proofs (although they recommend probabilistic proofs as more appropriate to their subject matter). Moreover, they implicitly concede the existence of classical bivalent proofs (insofar as the pursuit of probabilistic proofs may even lead to their discovery). They even go so far as to suggest the social processing of a probabilistic proof, like that of a traditional proof, can involve "enough internalization, enough transformation, enough generalization, enough use, . . . " that the mathematical community accepts it as correct: "The theorem is thought to be true in the classical sense—that is, in the sense that it could be demonstrated by formal, deductive logic, although for almost all theorems no such deduction ever took place or ever will" [8, pp. 273–274].

The force of their position appears to derive from the complexity that confronts those who would attempt to undertake mathematical proofs and program verifications. They report a formal demonstration of a conjecture by Ramanujan would require 2,000 pages to formalize. They lament that Russell and Whitehead "in three enormous, taxing volumes, (failed) to get beyond the elementary facts of arithmetic." These specific examples may be subject to dispute: for Ramanujan's conjecture, precisely how is such a fanciful estimate supposed to be derived and verified? For Russell and Whitehead, is *Principia Mathematica* therefore supposed to be a failure? And yet their basic point ("The lower bounds on the length of formal demonstrations for mathematical theorems are immense and there is no reason to believe that such demonstrations for programs would be any shorter or cleaner—quite the contrary") nevertheless merits serious consideration.

One of the most important ambiguities to arise within this context emerges at this juncture. While DeMillo, Lipton and Perlis [8, p. 278] suggest that the *scaling-up argument* (the contention that very complex programs and proofs can be broken down into much simpler programs and proofs for the purposes of verification and of demonstration) is the best the other side can produce, they want to deny it should be taken seriously insofar as it is supposed to depend upon an untenable assumption:

> The scaling-up argument seems to be based on the fuzzy notion that the world of programming is like the world of Newtonian physics—made up of smooth, continuous functions. But, in fact, programs are jagged and full of holes and caverns. Every programmer knows that altering a line or sometimes even a bit can utterly destroy a program or mutilate it in ways that we do not understand and cannot predict.

Indeed, since this argument is supposed to be the best argument in defense of the verificationist position, what they call "the discontinuous nature of programming" is said to sound "the death knell for verification."

Maurer has strenuously objected to their complaints that the verification of one program can never be transferred to any other program ("even to a program only one single line different from the original") and that there are no grounds to suppose that the verification of a large program could be broken down into smaller parts ("there is no reason to believe that a big verification can be the sum of many small verifications") [28, p. 278]. In response, he has observed that, while the modification of a correct program can produce an incorrect one for which "no amount of verification can prove it correct" and while minute changes in correct programs can produce "wildly erratic behavior . . . if only a single bit is changed," that does not affect the crucial result, namely: that proofs of the correctness of a program can be transferred to other programs, when those other programs are carefully controlled modifications of the original program, if not in whole then at least in part. Indeed, quite frequently, "if a program is broken up into a main program and n subroutines, we have $n + 1$ verifications to do, and that is all we have to do in proving program correctness."

CUMULATIVE COMPLEXITY AND PATCH-WORK COMPLEXITY

> *. . . verifications cannot be internalized, transformed, generalized, used, connected to other disciplines, and eventually incorporated into a community consciousness. They cannot acquire gradual credibility, as a mathematical theorem does; one either believes them blindly, as a pure act of faith, or not at all.*
>
> DeMillo, Lipton and Perlis

DeMillo, Lipton and Perlis, however, cannot be quite so readily dismissed, for their contentions on behalf of their position are quite intriguing:

> No programmer would agree that large production systems are composed of nothing more than algorithms and small programs. Patches, *ad hoc* constructions, Band-Aids and tourniquets, bells and whistles, glue, spit and polish, signature code, blood-sweat-and-tears, and, of course, the kitchen sink—the colorful jargon of the practicing programmer seems to be saying something about the nature of the structures he works with; maybe theoreticians ought to be listening to him. [8, p. 277]

Thus, it appears to be because most real software tends to consist of a lot of error messages and user interfaces—*ad hoc*, informal structures that are by definition unverifiable"—that they want to view the verificationist position as so far removed from the realities of programming life. But I think the strong arguments advanced on both sides of this issue suggest that two different concepts may be intimately intertwined that should be unraveled, where these involve differing dimensions of the nature of program complexity.

Maurer's position, for example, tends to support the view of complexity according to which more complex programs consist of less complex programs interacting according to some specific arrangement, a conception that is quite compatible with the scaling-up argument that DeMillo, Lipton and Perlis are inclined to disparage. This conception of complexity might be described as:

(C1) Cumulative Complexity =df
the complexity of larger programs arising when they consist of (relatively straightforward) arrangements of smaller programs;

as opposed to an alternative conception of complexity that is quite different:

(C2) Patch-Work Complexity =df
the complexity of larger programs arising when they consist of complicated, *ad hoc*, peculiar arrangements of smaller programs;

where the verificationist attitude appears to be appropriate to cumulative complexity, but far less adequate (perhaps hopelessly inappropriate) for cases of patch-work complexity, while the anti-verificationist attitude appears to have the opposite virtues in relation to cases of both of these kinds, respectively. Thus, the differences that distinguish large from small programs in the case of cumulative complexity tend to be differences of degree, whereas the differences that distinguish cumulative from patch-work complexity are differences in kind.[7]

If this reconstruction of the situation is approximately correct, then the verificationist approach, in principle, would appear to apply to small programs and to large programs when they exemplify cumulative complexity. However, it would not apply—or, at best, only in part—to those that exhibit patch-work complexity. Moreover, the *in principle* qualification is important here, because DeMillo, Lipton and Perlis offer reasons to doubt that large programs are or ever will be subject to verification, even when they are not "patch-work programs":

> The verification of even a puny program can run into dozens of pages, and there's not a light mo-

[7] Moreover, differences in degree with respect to measurable magnitudes—such as heights, weights, and so forth—are not precluded from qualifying as differences in kind. Indeed, to suppose that there cannot be any "real" difference between two things merely because there are innumerable intermediate degrees of difference between them is known as the fallacy of the continuum. The distinction recommended here is appropriate either way.

ment or a spark of wit on any of those pages. Nobody is going to run into a friend's office with a program verification. Nobody is going to sketch a verification out on a paper napkin. Nobody is going to buttonhole a colleague into listening to a verification. Nobody is ever going to read it. One can feel one's eyes glaze over at the very thought. [8, p. 276]

This enchanting passage is almost enough to beguile one into the belief that a program verification is among the most boring and insipid of all the world's objects. Yet even if that were true—even if it destroys all our prospects for a social process of program verification parallel to that found in mathematics—it would not establish the potential for their production as pointless and without value, if its purpose is properly understood.

Indeed, one of the most important insights that can be gleaned from reviewing this debate is an appreciation of the role of formal demonstrations in both mathematical and in programming contexts. For while it is perfectly appropriate for DeMillo, Lipton and Perlis to accentuate the historical truth that the vast majority of mathematical theorems and computer programs will never be subjected to the exquisite pleasures of formal validation or of program verification, it remains enormously important, as a theoretical possibility, that those theorems and programs could have been or could still be subjected to the critical scrutiny that such a thorough examination would provide. Indeed, from this point of view, the theoretical possibility of subjecting them to rigorous appraisal ought to be regarded as more important than its actual exercise. For it is this potentiality, whether or not it is actually exercised, that affords an objective foundation for the intersubjective evaluation of knowledge claims: z knows that something is the case only when z's belief that that is the case can be supported deductively, inductively, or whatever, not as something that has been done but as something that could be done were it required. This may be called "an examiner's view" of knowledge [14, p. 319].[8]

THEOREMS, ALGORITHMS, AND PROGRAMS

Lamport and Maurer display an amazing inability to distinguish between algorithms and programs.

DeMillo, Lipton and Perlis

The argument that has gone before, however, depends upon an assumption that DeMillo, Lipton and Perlis seem to be unwilling to grant: the presumption that programs are like theorems from a certain point of view. That some analogy exists between theorems and programs is a tempting inference, not least of all given their own analogy with mathematics. In Table I we begin with the most seemingly obvious comparison one might make. Thus, from this perspective, proofs are to theorems as verifications are to programs—demonstrations of their truth or correctness.[9] (To avoid ambiguity, we will assume the correctness of a program means "program correctness" rather than "specification correctness," even though, in other contexts, specification correctness is important as well. Our concern here is whether or not a program satisfies a certain set of specifications instead of whether those specifications are something a user may want to change at any point.)

TABLE I. A Plausible Analogy

	Mathematics	Programming
Objects of Inquiry:	Theorems	Programs
Methods of Inquiry:	Proofs	Verifications

This analogy, however, might not be as satisfactory as it initially appears. If we consider that programs can be viewed as functions from inputs to outputs, the features of mathematical proofs that serve as counterparts to inputs, outputs, and programs, respectively, are premises, conclusions, and rules of inference. Thus, in lieu of the plausible analogy with which we began, a more adequate conception emerges in Table II, where the acceptability of this analogy itself depends upon interpreting programs as well as rules of inference as functions (from a domain into a range). This analogy is still compatible with the first, however, so long as it is understood that rules of inference are used to derive theorems from premises, whereas programs are used to derive outputs from inputs. Thus, to the extent to which establishing that a certain theorem follows from certain premises (using certain rules of inference) is like establishing a certain output is generated by a certain input (by a certain program), proving a theorem is indeed like verifying a program.

TABLE II. A More Likely Analogy

	Mathematics	Programming
Domain:	Premises	Input
Function:	Rules of inference	Programs
Range:	Theorems	Output

In order to discern the deeper disanalogy between these activities, therefore, it is necessary to realize that while algorithms satisfy the conditions previously specified (of qualifying as functions from a domain to a range), programs need not do so. One reason, of course, emerges from considerations of complexity, especially when programs are patch-work complex programs,

[8] A fascinating illustration of this attitude may be found in Holt [21, pp. 24–25], who discusses what he takes to be "the three C's" of formal program specification: clarity, completeness, and consistency. He suggests that the formal verification of program correctness represents an idealization, "(much as certain laws of physics are idealizations of the motion of bodies of mass). This idealization encourages careful, proficient reasoning by programmers, *even if they choose not to carry out the actual mathematical steps in detail.*" Careful, proficient reasoning by programmers, however, is presumably attainable by methods other than program verification procedures.

[9] There are other aspects of specific programs, of course, including heuristics, especially, where the role of heuristics as "rules of thumb" or as guidelines that allow exceptions but are useful, nonetheless, could be the subject of further inquiry (emphasizing, for example, that discovering heuristics is an inductive enterprise). Here, however, the focus is upon the difference between programs and algorithms as objects of verification.

because programs of this kind have idiosyncratic features that make a difference to the performance of those programs, yet are not readily amenable to deductive procedures. The principal claim advanced by DeMillo, Lipton and Perlis (with respect to the issue here), is that patch-work complex programs have *ad hoc*, informal aspects such as error messages and user interfaces, that are unverifiable "by definition" [8, p. 227]. In cases of this kind, presumably, no axioms relating these special features of specific programs to their performance when executed are available, making verification impossible.

A more general reason, however, arises from features of other programs that are obviously intended to bring about the performance of special tasks by the machines that execute them. Illustrations of such tasks include the input and output behavior that is supposed to result as a causal consequence of their execution. Thus, the IBM PC manual for Microsoft BASIC defines various "I/O Statements" in the following fashion:[10]

(A)

Statement:	**Action:**
BEEP	Beeps the speaker.
CIRCLE (x, y) r	Draws a circle with center x, y and radius r.
COLOR b, p	In graphics mode, sets background color and pallette of foreground colors.
LOCATE row, col	Positions the cursor.
PLAY string	Plays music as specified by string.

Although these "statement" commands are expected to produce their corresponding "action" effects, it should not be especially surprising that prospects for their verification are problematic.

Advocates of program verification, however, might argue that commands like these are special cases. They are amenable to verification procedures, not in the sense of absolute verifiability, but rather in the sense of relative verifiability. The programs in which these commands appear could be subject to verification, provided special "causal axioms" are available relating the execution of these commands to the performance of the corresponding tasks. The third reason, therefore, ought to be even more disturbing for advocates of program verification. For it should be evident that even the simplest and most commonplace program implicitly possesses precisely the same causal character. Consider, for example, the following program, simple, written in Pascal:

(B)
```
program simple (output);
begin
  writeln ('2 + 2 =', 2 + 2);
end
```

This program, of course, instructs the machine to write "2 + 2 =" on a line followed by its solution "4" on the same line. For either of these outcomes to occur, however, obviously depends upon various different causal factors, including the characteristics of the compiler, the processor, the printer, the paper and every other component whose specific features influence the execution of such a program.

Taken all together, these considerations support the theoretical necessity to distinguish programs as encodings of algorithms from the logical structures that they represent. A program, after all, is a particular implementation of an algorithm in a form that is suitable for execution by a machine. In this sense, a program, unlike an algorithm, qualifies as a causal model of a logical structure of which a specific algorithm may be a specific instance. The consequences of this realization are enormous, insofar as causal models of logical structures need not have the same properties that characterize those logical structures themselves. Algorithms, rather than programs, thus appear to be the appropriate candidates for analogies with *pure mathematics*, while programs bear comparison with *applied mathematics*. Propositions in applied mathematics, unlike those in pure mathematics, run the risk of observational and experimental disconfirmation.

From this point of view, it becomes possible to appreciate why DeMillo, Lipton and Perlis accentuate the role of program testing as follows:

> It seems to us that the only potential virtue of program proving [verification] lies in the hope of obtaining perfection. If one now claims that a proof of correctness can raise confidence, even though it is not perfect or that an incomplete proof can help one locate errors, that that claim must be verified! There is absolutely no objective evidence that program verification is as effective as, say, *ad hoc* theory testing in this regard.

If not for the presumption that programs are not algorithms in some fundamental respects, these remarks would be very difficult—if not impossible—to understand. But when the assumption is made that,

(D1) algorithm =df
a logical structure of the type function suitable for the derivation of outputs when given inputs;

(D2) program =df
a causal model of an algorithm obtained by implementing that function in a form that is suitable for execution by a machine;

it is no longer difficult to understand why they should object to the conception of program verification as an inappropriate and unjustifiable exportation of a deductive procedure applicable to theorems and to algorithms for the purpose of evaluating causal models that are executed by machine.[11]

[10] I am grateful to my colleague, David Cole, for proposing these examples.

[11] The phrase "causal model," of course, has been bestowed upon entities as diverse as *scientific theories* (such as classical mechanics and relativity theory), *physical apparatus* (such as arrangements of ropes and pulleys), and even *operational definitions* (such as that IQs are what IQ tests test). Different disciplines tend to generate their own special senses. For discussion, see [15], [18], [34].

ABSTRACT MACHINES VERSUS TARGET MACHINES

I find digital computers of the present day to be very complicated and rather poorly defined. As a result, it is usually impractical to reason logically about their behavior. Sometimes, the only way of finding out what they will do is by experiment. Such experiments are certainly not mathematics. Unfortunately, they are not even science, because it is impossible to generalize from their results or to publish them for the benefit of other scientists.

C. A. R. Hoare

The conception of computer programs as causal models and the difference between programs and algorithms deserve elaboration, especially insofar as there are various senses in which something might or might not qualify as a program or as a causal model. The concept of a program is highly ambiguous, since the term "program" may be used to refer to (i) algorithms, (ii) encodings of algorithms, (iii) encodings of algorithms that can be compiled, or (iv) encodings of algorithms that can be compiled and executed by a machine. There are other program senses as well.[12] As an effective decision procedure, an algorithm is more abstract than is a program, insofar as the same algorithm might be implemented in different forms suitable for execution by various machines by using different languages. From this perspective, the senses of program defined by (ii), (iii) and (iv) provide conceptual benefits that the sense defined by (i) does not. Indeed, were "program" defined by sense (i), programs could not fail to be verifiable.

The second sense is of special importance within this context, especially in view of the distinction between "abstract machines" and "target machines." As noted earlier, source programs are written in high-level languages that simulate abstract machines, whose instructions can be more readily composed than can those of the target machines that ultimately execute them. It is entirely possible, in sense (ii), to envision the composition of a program as involving no more than the encoding of an algorithm in a programming language, no matter whether that program is now or ever will be executed by a machine or not. Indeed, it might be said that the composition of a program involves no more than the encoding of an algorithm in a programming language, even if that language cannot be executed by any machine at all. An instance of this state-of-affairs, moreover, is illustrated by the mini-language CORE, introduced by Marcotty and Ledgard [27] as a means for explaining the features characteristic of programming languages in general, without encountering the complexities involved in discussions of Pascal, LISP and so on. In cases of this kind, these languages may reflect properties of abstract machines for which there exist no actual target machine counterparts.

[12] A different set of five distinctive senses of "program" is suggested by [30]. Those cited here, however, appear appropriate for our purposes.

The crucial difference between programs in senses (i) and (ii) and programs in senses (iii) and (iv), therefore, is that (i) and (ii) can be satisfied merely by reference to abstract machines, whereas (iii) and (iv) require the existence of target machines. In the case of a mini-language like CORE, it might be argued that the abstract machine is the target machine. But this contention overlooks the difference at stake here because an abstract machine no more qualifies as a machine than an artificial flower qualifies as a flower. Compilers, interpreters, processors and the like are properly characterized as *physical things*, i.e., as systems in space/time for which causal relations obtain. Abstract machines are properly characterized as *abstract entities*, i.e., as systems not in space/time for which only logical relations obtain. It follows that, in senses (i) and (ii), the intended interpretations of programs are abstract machines that are not supposed to have physical machine counterparts. But in senses (iii) and (iv), the intended interpretations of programs are abstract machines that are supposed to have physical machine counterparts. And this difference is crucial: it corresponds to the intended difference between definitions (D1) and (D2).

On the basis of these distinctions, it should be evident that algorithms—and programs in senses (i) and (ii)—are subject to absolute verification by means of deductive procedures. This possibility occurs because the properties of abstract machines that have no physical machine counterparts can be established by definition, i.e., through stipulations or conventions, which might be formalized either by means of program rules of inference or by means of primitive program axioms. In this sense, the abstract machine under consideration simply *is* the abstract entities and relations thereby specified. By comparison, programs in senses (iii) and (iv) are merely subject to relative verification, at best, by means of deductive procedures. Their differences from algorithms arise precisely because, in these cases, the properties of the abstract machines they represent, in turn, stand for physical machines whose properties can only be established inductively. With programs, unlike algorithms, there are no "program rules of inference" or "primitive program axioms" whose truth is ascertainable by definition.

In either case, however, all these rules and axioms relate the occurrence of an input I to the occurrence of an output O, which can be written in the form of claims that input I causes output O more or less as follows:

(C) $\qquad I \; c \; O;$

thus, given this rule or axiom, the occurrence of output O may be inferred from the occurrence of input I together with a rule or axiom of that form:

(D) From "$I \; c \; O$" and "I" infer to "O";

thus, from axiom or rule "$I \; c \; O$" and input "I", output "O" validly follows. The difference between algorithms

and programs, from this point of view, is that patterns of reasoning of form (D) are absolutely verifiable in the case of algorithms but are only relatively verifiable in the case of programs—a difference that reflects the fact that claims of form (C) can be established by deductive procedures as definitional stipulations with respect to algorithms, but can only be ascertained by inductive procedures, as lawful or causal generalizations, in the case of programs, thus understood.

THE VERY IDEA OF PROGRAM VERIFICATION

> *When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics.*
>
> C. A. R. Hoare

When entertained from this point of view, the fundamental difficulty encountered in attempting to apply deductive methodology to the verification of programs does not appear to arise from either the idiosyncracy of various features of those programs or from the inclusion of instructions for special tasks to be performed. Both types of cases can be envisioned as matters that can be dealt with by introducing special rules and special axioms that correspond to either the *ad hoc* features of those patch-work complex programs or to the special behavior that is supposed to be exhibited in response to special commands. The specific inputs "*I*" and the specific outputs "*O*", after all, can be taken to cover these special kinds of cases. Let us therefore take this for granted in order to provide the strongest case possible for the verificationist position and thereby avoid any chance of being charged with having attacked a straw man.

As we discovered, the crucial problem confronting program verification is establishing the truth of claims of form (C) above, which might be done in two possible ways. The first is to interpret rules and axioms of form (C) as definitional truths concerning the abstract machine thereby defined. The other is to interpret rules and axioms of form (C) as empirical claims concerning the possible behavior of the target machine thereby described. Interpreted in the first way, the performance of an abstract machine can be conclusively verified, but it possesses no significance at all for the performance of any physical system. Interpreted in the second way, the performance of an abstract machine possesses significance for the performance of a physical system, but it cannot be conclusively verified. And the reason, by now, should be obvious; for programs are subject to "relative" rather than "absolute" verification, in relation to "rules and axioms" in the form of lawful and causal generalizations as premises—empirical claims whose truth can never be established with certainty!

The very idea of program verification trades upon an equivocation. Interpreted in senses (i) and (ii), there is no special difficulty that arises in "verifying" that output *O* follows from input *I* as a logical consequence of axioms of the form, *I c O*. Under such an interpretation, however, nothing follows from the verification of a "program" concerning the performance of any physical machine. In this case, the absolute verification of an abstract machine is theoretically possible and is not particularly problematic. Interpreted in senses (iii) and (iv), however, that output *O* follows from input *I* as a logical consequence of axioms of the form, *I c O*, cannot be subject to absolute verification, precisely because the truth of these axioms depends upon the causal properties of physical systems, whose presence or absence is only ascertainable by means of inductive procedures. In this case, the absolute verification of an abstract machine is logically impossible because its intended interpretation is a target machine whose behavior might not be described by those axioms, whose truth can only be established by induction.

This conclusion strongly suggests the conception of programming as a mathematical activity requires qualification in order to be justified. For while it follows from the axioms for the theory of natural numbers that, say,

(E) $$2 + 2 = 4;$$

the application of that proposition—which may be true of the abstract domain to which its intended interpretation refers—for the purpose of describing the causal behavior of physical things like alcohol and water need not remain true:

(F) 2 units of water + 2 units or alcohol = 4 units of mixture.

For while the abstract proposition (E) is true, the empirical proposition (F) is false. The difference involved here is precisely that between *pure* mathematics and *applied* mathematics.[13] When the function of a program is merely to satisfy the constraints imposed by an abstract machine for which there is no intended interpretation with respect to any physical system, then the behavior of that system can be subject to conclusive absolute verification. This scenario makes Hoare's [20] four basic principles true because then:

(1) computers are mathematical machines;
(2) computer programs are mathematical expressions;
(3) a programming language is a mathematical theory; and,
(4) programming is a mathematical activity.

But if the function of a program is to satisfy the constraints imposed by an abstract machine for which there is an intended interpretation with respect to a physical system, then the behavior of that system cannot be subject to conclusive absolute verification but

[13] On the difference between pure and applied mathematics, see especially Hempel [16, 17] which are as relevant today as they were then.

requires instead empirical inductive investigation to support inconclusive relative verifications. In cases of this kind, Hoare's four principles are false and require displacement as follows:

(1) computers are applied mathematical machines;
(2) computer programs are applied mathematical expressions;
(3) a programming language is an applied mathematical theory; and,
(4) programming is an applied mathematical activity;

where propositions in applied mathematics, unlike those in pure mathematics, run the risk of observational and experimental disconfirmation.

COMPUTER PROGRAMS AS APPLIED MATHEMATICS

A geometrical theory in physical interpretation can never be validated with mathematical certainty, no matter how extensive the experimental tests to which it is subjected; like any other theory of empirical science, it can acquire only a more or less high degree of confirmation.

C. G. Hempel

The differences between pure and applied mathematics are very great, indeed. As Einstein remarked, insofar as the laws of mathematics refer to reality, they are not certain; and insofar as they are certain, they do not refer to reality. DeMillo, Lipton and Perlis likewise want to maintain that, to the extent to which verification has a place in programming practice, it applies to the evaluation of algorithms; and to the extent to which programming practice goes beyond the evaluation of algorithms, it cannot rely upon verification. Indeed, from the perspective of the classical theory of knowledge, their position makes excellent sense; for the investigation of the properties of programs (thus understood) falls within the domain of inductive methodology, while the investigation of the properties of algorithms falls within the domain of deductive methodology. As we have discovered, these are not the same.

Since the behavior of algorithms can be known with certainty (within the limitations of deductive procedure), but the behavior of programs can only be known with uncertainty (within the limitations of inductive procedure), the degree of belief (or "strength of conviction") to which specific algorithms and specific programs are entitled can vary greatly. In particular, a hypothetical scale once again may be constructed, where, to the extent to which a person can properly claim to be rational with respect to his beliefs, there should be an appropriate correspondence (which need not necessarily be an identity) between his degree of subjective belief that something is the case (as displayed by Figure 2) and the measure of objective evidence in its support (as displayed by Figure 1). Otherwise, such a person does not distribute those degrees of

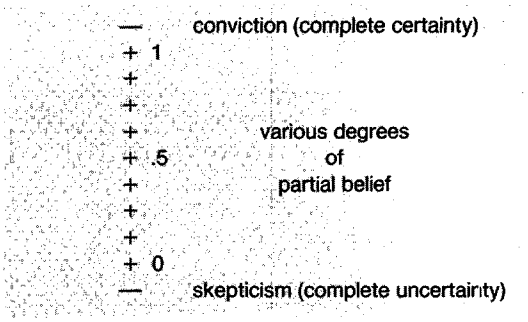

FIGURE 2. A Measure of Subjective Belief

belief in accordance with the available evidence and is to that extent irrational [11, ch. 10].

The conception of a program as a causal model suitable for execution by a machine reflects the interpretation of programs as causal factors that interact with other causal factors to bring about a specific output as an effect of the introduction of specific input. As everyone appears willing to admit, the execution of a program qualifies as causally complex, insofar as even a correct program can produce "wildly erratic behavior . . . if only a single bit is changed." The reason the results of executing a program cannot provide deductive support for the evaluation of a program [as Hoare 20, p. 116 acknowledges], moreover is that the behavior displayed by a causal system is an effect of the complete set of relevant factors whose presence or absence made a difference to its production. Indeed, this reflection has an analog with respect to inductive procedure, since it is a fundamental principle of inductive methodology that measures of evidential support should be based upon the complete set of relevant evidence that is currently available, where any finding whose truth or falsity increases or decreases the evidential support for a conclusion is evidentially relevant.[14]

The principle of maximal specificity for causal systems and the principle of total evidence for inductive methodology are mutually reinforcing: reasoning about programs tends to be (a) non-demonstrative, (b) ampliative, and (c) non-additive precisely because the truth of a generalization about a causal system depends upon *its complete specification*, which can be very difficult—if not practically impossible—to obtain. In the absence of information of this kind, however, the best knowledge available is uncertain. Indeed, if the knowledge that deductive warrants can provide is said to be "perfect," then our knowledge of the behavior of causal systems must always be "imperfect," experimental and tentative (like physics) rather than demonstrative and certain (like mathematics). It is therefore ironic to discover the position advanced by DeMillo, Lipton and Perlis implic-

[14] That every property whose presence or absence makes a difference to the performance of a causal system must be taken into account in order to understand its behavior is known as *the requirement of maximal specificity*; that inductive reasoning must be based upon all the available relevant evidence is known as *the requirement of total evidence*. [11].

itly entails "the fuzzy notion that the world of programming is like the world of Newtonian physics"—not in its subject matter, of course, but in its methodology.

INDUCTIVE TESTING AND COMPUTER PROGRAMS

Could the God that plays dice trigger a nuclear holocaust by a random error in a military computer?
H. A. Pagels

At least two lines of defense might be advanced against this conclusion, one of which depends upon the possibility of an *ideal* programmer, the other upon the prospects for verification by *machine*. The idea of an ideal programmer is that of a programmer who knows as much about algorithms, programming and computers as there is to know. When this programmer is satisfied with a program, by hypothesis that program is "correct." The catch is two-fold. First, how could any programmer possess the knowledge required to be an "ideal programmer"? Unless this person were God, we may safely assume the knowledge he possesses has been ascertained by means of the usual methods, including (fallible) inductive reasoning about the future behavior of complex systems based upon evidence about their past behavior. Second, even a correct program is but one feature of a complex causal system. The performance of a computer while executing a program depends not only upon the software but also upon the firmware, the hardware, the input/output devices and all the rest. While it would not be mistaken to suggest that, *ceteris paribus*, if these other components perform as they should, the system will perform as it should, such claims are not testable.

The emphasis here is on the word "should." Since the outputs that result from various inputs are complex effects of an interacting arrangement of software, firmware, hardware, and so on, the determination that some specific component of such a system functions properly ("as it should") depends upon assumptions concerning the specific states of each of the others, in the absence of which, strictly speaking, no program as such can be subject to test. Even when the specific states of the relevant components have been explicitly specified, the production of output O given input I on one occasion provides no guarantee that output O would be produced by input I on another occasion. Indeed, the type of system created by the interaction of these component parts could be probabilistic rather than deterministic. Repeated tests of any such system can provide only inductive evidence of reliability. Taken together, these considerations suggest that even the idea of an ideal programmer cannot improve the prospects for program verification.[15]

[15] An alternative position could emphasize the social processing whereby more than one "very good programmer" might come to accept or to believe in the correctness of a program. As we have already discovered, this will not do. After all, the claim, "This program reflects what programmer z believes are the right commands," could be true, when the sentence, "This program reflects the right commands," happens to be false. The first is a claim about z's *beliefs*, the latter about a *program*—even when z is a programming group.

A fascinating example of the kind of difficulty that can be encountered is illustrated by the distinction between "hard" and "soft" errors, which occur when quantum phenomena are implicated in situations like these:

> In the 1980s a new generation of high-speed computers will appear with switching devices in the electronic components which are so small they are approaching the molecular microworld in size. Old computers were subject to "hard errors"—a malfunction of a part, like a circuit burning out or a broken wire, which had to be replaced before the computer could work properly. But the new computers are subject to a qualitatively different kind of malfunction called "soft errors" in which a tiny switch fails during only one operation—the next time it works fine again. Engineers cannot repair computers for this kind of malfunction because nothing is actually broken. [31, p. 125]

It should be clear by now that the difference between proving a theorem and verifying a program does not depend upon the presence or absence of any social process during their production, but rather upon the presence or absence of causal significance. A summary of their parallels is illustrated in Table III. The chart shows the difference in social processes is merely a difference in practice, but the difference in causal significance is a real difference in principle.

TABLE III. A Final Comparison

	Proving theorems	Verifying programs
Syntactic objects of inquiry:	Yes	Yes
Social process of production:	Yes	No
Physical counterparts:	No	Yes
Causal significance:	No	Yes

The suggestion can also be made that program verification might be performed by higher-level machines that have the capacity to validate proofs that are many orders of magnitude more complex than those that can be mechanically verified today. This possibility implicitly raises issues of mentality concerning whether or not computers have the capacity for semantic interpretation as well as for syntactic manipulation [12]. Even assuming that their powers are limited to string processing, this is an intriguing prospect, especially since proofs and programs alike are syntactic entities. There are no special difficulties so long as their intended interpretations are abstract machines. When their intended interpretations are target machines, then we encounter the problem of determining the reliability of the verifying programs themselves ("How do we verify the verifiers?"), which invites a regress of relative verifications of relative verifications. As long as our reasoning is valid, causally significant conclusions can be derived only from causally significant premises.

Tymoczko [36], however, suggests that proof proce-

dures in mathematics have three distinctive features: they are convincing, they are formalizable and they are surveyable. His sense of "surveyability" can be adequately represented by "replicability." Proofs are convincing to mathematicians because they can be formalized and replicated. Insofar as the machine verification of programs has the capacity to satisfy the desiderata of formalizability and of replicability, their successfully replicated results ought to constitute evidence for their correctness.[16] This, of course, does not alter the inductive character of the support thereby attained nor does it increase the range of the potential application of program verification, but it would be foolish to doubt the importance of computers in extending our reasoning capabilities, just as telescopes, microscopes, and all the rest have extended our sensible capacities: verifying programs, after all, could be published (just as proofs are published) in order to be subjected to the criticism of the community—even by means of computer trials! Even machine verifications, however, cannot guarantee the performance that will result from executing a program, which is yet one more form of our dilemma.[17]

COMPLEXITY AND RELIABILITY

> *We must therefore come to grips with two problems that have occupied engineers for many generations: First, people must plunge into activities that they do not understand. Second, people cannot create perfect mechanisms.*
>
> DeMillo, Lipton and Perlis

From this perspective, the admonitions advanced by DeMillo, Lipton and Perlis against the pursuit of perfection when perfection cannot be realized are clearly telling in this era of dependence upon technology. 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. When they assess the situation with respect to critical cases (such as air-traffic control, missile systems, and the like) in which human lives are at risk, the ominous significance of their position appears to be overwhelming:

> . . . the stakes do not affect our belief in the basic impossibility of verifying any system large enough and flexible enough to do any real-world task. No matter how high the payoff, no one will ever be able to force himself to read the incredibly long, tedious verifications of real-life systems, and, unless they can be read, understood, and refined, the verifications are worthless.

Thus, even when allowance is made for the possibility of group collaboration, the mistaken assumption that program performance can be guaranteed could easily engender an untenable conception of the situation encountered when human lives are placed in jeopardy as in the case of SDI. If one were to assume the execution of a program could be anticipated with the mathematical precision that is characteristic of demonstrative domains, then one might more readily succumb to the temptation to conclude that decisions can be made with complete confidence in the (possibly unpredictable) operational performance of a complex causal system.

Complex systems like SDI are heterogeneous arrangements of complicated components, many of which combine hardware and software. They depend upon sensors providing real-time streams of data where, to attain rapid and compact processing, their avionics portions are programmed in assembly language—a type of programming that does not lend itself to the construction of program verifications. Systems like these differ in important respects from, say, brief programs that read an ASCII file in order to count the number of characters or words per line. Any analysis of computer-system reliability that collapses these diverse types of systems and programs into a single catch-all category would be indefensibly oversimplified. Therefore it should be emphasized that, in addition to the difference between absolute and relative verifiability with respect to programs themselves, the reliability of inductive testing—not to mention observation techniques—declines as the complexity of the system increases. As a rule of thumb, the more complex the system, the less likely it is to perform as desired [22]. The operational performance of these complex systems should never be taken for granted and cannot be guaranteed.

The blunder that would be involved in thinking otherwise does not result from the presence or the absence of social processes within this field that might foster the criticism of a community, but emanates from the very nature of these objects of inquiry themselves. The fact that one or more persons of saintly disposition might sacrifice themselves to the tedium of eternal verification of tens of millions of lines of code for the benefit of the human race is beside the point. The limitations involved here are not merely practical: *they are rooted in the very character of causal systems themselves.* From a methodological point of view, it might be said that programs are conjectures, while executions are attempted—and all too frequently successful—refutations (in the spirit of Popper [32, 33]). Indeed, the more serious the consequences that would attend the making of a mistake, the greater the obligation to insure that it is not made. In maintaining that program verification cannot succeed as a generally applicable and completely reliable method for guaranteeing the performance of a program, DeMillo, Lipton and Perlis thus arrived at the right general conclusion for the wrong specific reasons. Still, we are all indebted to them for their efforts to clarify a confusion whose potential con-

[16] For an intriguing discussion of these questions in relation to the proof-by-computer of the four-color problem, see [9], [35] and [36].

[17] As Cerutti and Davis [5, pp. 903–904], have observed, "For machine proofs, we can (a) run the program several times, (b) inspect the program, (c) invite other people to inspect the program or to write and run similar programs. In this way, if a common result is repeatedly obtained, one's degree of belief in the theorem goes up". An interesting discussion of their position may be found in [9, esp., pp. 805–807]; but Detlefsen and Luker beg the question in assuming that people are simply a different kind of computer, a crucial question on which they shed no light.

sequences—not only for the community of computer science, but for the human race—cannot be overstated and had best be understood.

Acknowledgments. The original version of this paper was composed while a Post-Doctoral Fellow in computer science at Wright State University. Special thanks to Henry Davis, David Hemmendinger, Jack Kulas and especially Al Sanders for encouraging a philosopher to poke around in their backyard. I am also indebted to two very conscientious but anonymous referees for this magazine and to Rob Kling and Chuck Dunlop for their stimulating criticism and penetrating inquiries, which forced me to clarify the basis for my position.

REFERENCES

1. Benacerraf, P. and Putnam, H., Eds. *Philosophy of Mathematics: Selected Readings.* Prentice-Hall, Englewood Cliffs, N.J. 1964.
2. Black, M. Induction. *The Encyclopedia of Philosophy*, vol. 4, Edwards, P., Editor-in-Chief. Macmillan, New York, 1967, pp. 169–181.
3. Blumberg, A. Logic, modern, *The Encyclopedia of Philosophy*, vol. 5, Edwards, P., Editor-in-Chief, Macmillan, New York, 1967, pp. 12–34.
4. Bochner, S. *The Role of Mathematics in the Rise of Science.* Princeton Univ. Press, Princeton, N.J. 1966.
5. Cerutti, E. and Davis, P. Formac meets Pappus. *Am. Math. Monthly 76* (1969), 895–904.
6. Church, A. Logistic system. *Dictionary of Philosophy.* Runes, D., Ed. Littlefield, Adams & Co., Ames, Iowa, 1959, pp. 182–183.
7. Dancy, J. *An Introduction to Contemporary Epistemology.* Blackwell, Oxford, 1985.
8. DeMillo, R., Lipton, R. and Perlis, A. Social processes and proofs of theorems and programs. *Commun. ACM 22*, 5 (May 1979), 271–280.
9. Detlefsen, M. and Luker, M. The four-color theorem and mathematical proof. *J. Philos. 77*, 12 (December 1980), 803–820.
10. Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
11. Fetzer, J. H. *Scientific Knowledge.* Reidel, Dordrecht, Holland, 1981.
12. Fetzer, J. H. Signs and minds: An introduction to the theory of semiotic systems. In *Aspects of Artificial Intelligence*, Fetzer, J., Ed. Kluwer, Dordrecht/Boston/London/Tokyo, 1988, pp. 133–161.
13. Glazer, D. Letter to the editor. *Commun. ACM 22*, 11 (November 1979), 621.
14. Hacking, I. Slightly more realistic personal probabilities. *Philos. Sci. 34*, 4 (December 1967), 311–325.
15. Heise, D. R. *Causal Analysis.* Wiley, New York, 1975.
16. Hempel, C. G. On the nature of mathematical truth. In *Readings in Philosophical Analysis*, Feigl, H. and Sellars, W., Eds. Appleton-Century-Crofts, New York, 1949, pp. 222–237.
17. Hempel, C. G. Geometry and empirical science. In *Readings in Philosophical Analysis*, Feigl, H. and Sellars, W., Eds. Appleton-Century-Crofts, New York, 1949, pp. 238–249.
18. Hesse, M. *Models and Analogies in Science.* Univ. of Notre Dame Press, Notre Dame, Ind., 1966.
19. Hoare, C. A. R. An axiomatic basis for computer programming. *Commun. ACM 12* (1969), 576–580, 583.
20. Hoare, C. A. R. Mathematics of programming. *BYTE* (August 1986), 115–149.
21. Holt, R. Design goals for the Turing programming language. Technical Report CSRI-187 (Aug. 1986), Computer Systems Research Institute, Univ. of Toronto.
22. Kling, R. Defining the boundaries of computing across complex organizations. In *Critical Issues in Information Systems.* Boland, R. and Hirschheim, R. (Eds.). Wiley, New York, 1987.
23. Kuhn, T. S. *The Structure of Scientific Revolutions*, 2d ed. Univ. of Chicago Press, Chicago, 1970.
24. Lakatos, I. *Proofs and Refutations.* Cambridge Univ. Press, Cambridge, U.K., 1976.
25. Lakatos, I., and Musgrave, A., Eds. *Criticism and the Growth of Knowledge.* Cambridge Univ. Press, Cambridge, U.K., 1970.
26. Lamport, L. Letter to the editor. *Commun. ACM 22*, 11 (November 1979), 624.
27. Marcotty, M. and Ledgard, H. *Programming Language Landscape: Syntax/Semantics/Implementations*, 2d ed. Science Research Associates, Chicago, 1986.
28. Maurer, W. D. Letter to the editor, *Commun. ACM 22*, 11 (November 1979), 625–629.
29. Michalos, A. *Principles of Logic.* Prentice-Hall, Englewood Cliffs, N.J., 1969.
30. Moor, J. H. The pseudorealization fallacy and the Chinese room. In *Aspects of Artificial Intelligence.* Fetzer, J. Ed. Kluwer, Dordrecht/Boston/London/Tokyo, 1988, pp. 35–53.
31. Pagels, H. *The Cosmic Code.* Simon & Schuster, New York, 1982.
32. Popper, K. R. *Conjectures and Refutations.* Harper & Row, New York, 1965.
33. Popper, K. R. *Objective Knowledge.* Clarendon Press, Oxford, 1972.
34. Suppe, F., Ed. *The Structure of Scientific Theories*, 2d ed. University of Illinois Press, Urbana, Ill., 1977.
35. Teller, P. Computer proof. *J. Philos. 77*, 12 (December 1980), 797–803.
36. Tymoczko, T. The four-color theorem and its philosophical significance. *J. Philos. 76*, 2 (February 1979), 57–83.
37. van den Bos, J. Letter to the editor. *Commun. ACM 22*, 11 (November 1979), 623.

CR Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Methodologies; D.2.4 [**Software Engineering**]: Validation; D.2.5 [**Testing and Debugging**]: Diagnostics; F.2.2 [**Nonnumerical Algorithms and Problems**]: Complexity of Proof Procedures; F.3.1 [**Logics and Meaning of Programs**]: Mechanical Verification

General Terms: Algorithms, Experimentation, Human Factors, Performance, Reliability, Verification

Additional Key Words and Phrases: Causal models, deductive reasoning, inductive reasoning, logical structures, probabilistic verifications

James H. Fetzer, professor of Philosophy at the University of Minnesota, Duluth, is the series editor of *Studies in Cognitive Systems*, a new professional library published by Kluwer. He recently authored *Probability and Causality* (1988) and *Aspects of Artificial Intelligence* (1988). Author's present address: James H. Fetzer, Department of Philosophy and Humanities, University of Minnesota, Duluth, MN 55812.