CMU-CS-83-142

# First Steps Towards Inferential Programming

*William L. SCHERLIS and Dana S. SCOTT*

Department of Computer Science, Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213, USA

July 1983

**Abstract.** Logics of programs, while they have contributed significantly to our understanding of individual programs and to our knowledge of programming language design, have had disappointingly little influence on the methods by which programs are constructed and documented in practice. The reason for this, we suspect, is that the understanding embodied in these systems deals with individual programs and does not directly address the process by which programs are constructed. By focusing attention on this process, attempting to discern the fundamental steps in the evolution of programs, we propose that it may be possible to develop a logical system—supported by an appropriate machine environment—that will be more directly applicable to programming practice. The benefits of such a point of view will be discussed.

This is a slightly revised version of a paper to appear in the proceedings of the IFIP 83 Congress.

This research was supported in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539, and in part by the U.S. Army Communications R&D Command under Contract DAAK80-81-K-0074. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Our basic premise is that the ability to construct and modify programs will not improve without a new and comprehensive look at the entire programming process. Past theoretical research, say, in the logics of programs, has tended to focus on methods for reasoning about individual programs; little has been done, it seems to us, to develop a sound understanding of the *process* of programming—the process by which programs evolve in concept and in practice. At present, we lack the means to describe the techniques of program construction and improvement in ways that properly link verification, documentation and adaptability.

The attitude that takes these factors and their dynamics into account we propose to call "inferential programming." The problem of developing this attitude and the tools required is far from easy and needs extensive investigation, and in this paper we are only going to be able to discuss it in rather broad terms. We wish to suggest, in particular, two goals for research that can build on past experience but that enter at what we feel is the right level of generality. This is the topic of Section 1. In Section 2 we set forth our views on the conceptualization of the notion of program derivation and the reasons why we think it is needed. The discussion inevitably brings up the question of the rôle of formalization, which is the subject of Section 3. In Section 4 we try to lay out the difficulties of the inferential programming problem; while in Section 5 we speculate briefly on the future of programming.

## 1. A Research Programme.

The initial goal of research must be to discern the basic structural elements of the process of programming and to cast them into a precise framework for expressing them and reasoning about them. This understanding must be embodied in a *logic of programming*—a mathematical system that allows us not only to reason about individual programs, but also to carry out *operations* (or transformations) on programs and, most importantly, to reason about the operations. It can be argued—not without controversy—that logicians did this service for mathematical proof, and we will discuss the *pros* and *cons* presently when we argue that programming is even more in need of this kind of attention. The development of such a logic of programming will of course require a practical understanding of the semantics of programming concepts. There is no reason to suppose that the formalization of this logic will look like standard existing systems, since the principles must be adapted to the motivating problem and the appropriate concepts.

The second goal for research is to build the experimental environment. The logic of programming is intended to be a "systematized" one, codifying our competence about programming in a way that can be used as the basis for an implemented computer system. It will be necessary to construct prototype *interactive* systems to facilitate the exploration of logics of programming and, eventually, to lead us to the natural development of practical semantically-based programming tools. In particular, it is important that such systems will permit study of the *dynamics* of programming: the relation between the way in which derivations of programs from specifications are structured conceptually and the process, over time, by which they actually evolve and are maintained and modified. This dynamical aspect we find lacking in current proposals.

This programme of research is aimed at discovering the principles that we feel can be embodied in the programming tools of the future. Such principles must be independent of

individual programs—or even programming languages—if they are going to have reasonably universal significance. By developing conceptual and mechanical tools for expressing these principles and for reasoning with them, we hope to demonstrate that programming can be made a more straightforward and exact process and that the power of the programmer to think directly about the problems he has to solve can be significantly increased.

In thinking about the problem, we have found that it is important to distinguish program derivation—the *conceptual* history of a program—from what we have called *inferential programming*, by which we mean the collection of methods (and associated tools) for constructing, modifying, and reasoning about such derivations over time. Let us discuss this distinction in more detail.

**Program Derivation.** Contemporary programming languages and methodologies, we claim, encourage programmers to try to capture an entire process of program development in the *text* of a single program; programmers find they are attempting to write programs that—in themselves—can be easily understood and modified and yet have acceptable performance properties. Inevitably, there must be sacrifices in order to obtain the right balance between clarity and efficiency; often, perhaps more often than not, the greater sacrifice is from clarity, and the resulting programs become so complex and interconnected that eventual modification becomes prohibitively costly.

Many programmers feel it is more natural to describe a program in terms of its *derivation* or *evolution*--the sequence of insights that is required to derive the implementation from straightforward specifications. By representing the process of program development as a *sequence* of programs, arranged as if the final implementation were developed from an initial specification by a series of refinement steps, we can maintain a structure in which clarity and efficiency coexist. Separations between program abstractions (such as abstract data types or generic procedures) and their representations do not exist *within* individual programs in derivations, but rather are spread over a sequence of program derivation steps. Abstractions introduced in early derivation steps are replaced, in later steps, by their intended representations, allowing more specialized and, hence, faster code to be ultimately obtained. The programmer need never confront the possibility of having to maintain the abstraction *and* an efficient implementation simultaneously in a single program.

Programming, even more so than mathematics, is a highly stylized affair, in which certain patterns of activity are shared in large numbers of applications. This, indeed, is an argument that programming is largely a *skill*; the good programmers are not only smarter, but they have command of a larger collection of standard programming patterns. Although attempts have been made to describe these patterns in terms of the program *text* through which they are made manifest, we believe that the patterns are really patterns of *derivation*, and the textual similarities are only superficial.

**Inferential Programming.** Inferential programming, on the other hand, is like the *process* of building mathematical proofs: Mathematicians do not develop proofs by starting at line one and considering their possible moves from there. Rather, they formulate a strategy and fill in gaps until they have enough detail to make a convincing argument. The proof text that emerges is a highly structured justification for a mathematical fact—even

if it is written in ordinary language. The *process* of building the proof, on the other hand, is a somewhat undisciplined and exploratory activity, in which insights are gained and expressed and finally woven together to form a mathematical argument. By analogy, then, we prefer to separate program derivations—highly structured justifications for programs—from inferential programming—the process of building, manipulating, and reasoning about program derivations.

It follows that in proposing structure for program derivations, we are in no way attempting to coerce programmers to follow a specific temporal discipline in building their programs. We intend, rather, that they be provided with inferential programming *tools*—conceptual and mechanical aids—to facilitate the expression of the program and its justification and to help in the process of program development. In an inferential programming framework programmers can focus their thoughts less on expressing actual programs, and more on expressing *how* the programs come about. As a consequence, we claim, programming language design ceases to be the critical issue in programming methodology. It is likely that ultimately conventional programming languages will be required only for the very last refinement steps—and only because these steps precede a conventional compilation step.

As regards verification, inferential programming will offer a more natural method for proving correctness of complicated programs than do conventional techniques. The common approach to program proof has been to develop program and specification first, and then prove correctness as a separate step. The tediousness and difficulty of these proofs has prompted much of the software engineering community to abandon hope for the economic development of provably correct programs. Indeed, this style of proof requires programmers to rediscover the insights that went into the original development of the program and express them in a formal logical language. Unlike conventional approaches to correctness, inferential programming techniques—particularly when embodied in mechanical tools—will effectively allow programs and proofs to be developed simultaneously. By representing programs as sequences of derivation steps and using systematic techniques to move from one step to the next, correctness of the final program follows directly from the correctness of the derivation techniques. With cleverly constructed mechanical tools, the business of proof could be so effectively hidden from users that the development of correctness proofs seems to be automatic. Of course, those steps in a derivation whose correctness has not been proven can be isolated to facilitate debugging and testing.

We must emphasize that the idea of controlling program derivation is by no means new, and there has already been considerable activity (take [Balzer81, Bauer82, Cheatham72, Cheatham79, Feather82, Green81, Schwartz77, Wile81], just to name a few). These groups have found that, in the process of trying to build advanced program development tools and heuristic systems for reasoning about programs, it was very difficult to reason about the structure and meaning of programs within a purely program-oriented framework and, instead, some sort of evolutionary transformational paradigm must be used. A variety of systems have emerged, but they all seem to have this aspect in common. The experience has taught us much about the structure of programs and the methods by which they are developed, though much of this work has been misinterpreted, we feel. There is still confusion on certain important points, and it is our aim here to correct some misconceptions

we have perceived about where this research is going and to suggest some directions for future work.

## 2. The Nature of Program Derivations.

The traditional "correctness" proof—that a program is consistent with its specifications—does not constitute a derivation of the program. Conventional proofs, as currently presented in the literature, do little to justify the *structure* of the program being proved, and they do even less to aid in the development of new programs that may be similar to the program that was proved. That is, they neither *explicate* existing programs nor *aid* in their modification and adaptation.

We intend that program derivations serve as conceptual or idealized histories of the development of programs. That is, a program derivation can be considered an idealized record of the sequence of design decisions that led to a particular realization of a specification. It is not a true history of the discovery of the program in that it does *not* include the usual blind alleys and false starts, and it does not reveal how the initial specifications were actually arrived at. But it does, nevertheless, show how the shape of the final implementation is determined by a combination of design decisions and commitments. The importance of choosing the right level of abstraction is substantiated by consideration of the necessary *changes* that have to be made in programs when new needs are imposed.

**Modification and Adaptability.** The constructive quality of program derivations is exactly what makes them particularly useful in environments in which programs eventually must be adapted to other uses. Indeed, a very important advantage that we see coming from the inferential programming technique will concern program modification—an activity which reportedly demands the largest proportion of available programmer time in industry and government today.

It is common wisdom that in many circumstances it is better to modify an old program than to develop an entirely new program. This is clearly appropriate when the developments of the old and new programs would have much in common—in our terms, when there would be significant sharing in the program derivations. This can be the case even if the resulting target programs have little in common. Modification is difficult in a conventional framework because, like *a posteriori* verification, it requires rediscovery of concepts used during development of an implementation. Simple conceptual changes to a specification often require complicated and extensive changes to code. In an inferential programming system, not only can the conceptual changes be made directly at the appropriate places in program derivations, but the supporting system can be used to help propagate these changes correctly into implementations.

This kind of adaptability is important not only in the broad "software engineering" context, but in the development of localized fragments of code as well. Much of current programming practice consists of adapting general algorithms and techniques from textbooks or other programs to particular applications. Abstraction mechanisms in languages can alleviate much of this problem, by permitting a single generalized template of an algorithm or other programming abstraction to serve in many contexts. There are many cases,

however, in which the application required does not match a true instance of the template developed. In these cases, the connection between original algorithm and intended use is one of *analogy*, and a much more sophisticated mechanism than simple instantiation is required to establish the connection. Because the program derivation reveals so much more about the structure of programs, we believe that patterns of analogy are more likely to be apprehended and expressed when derivations are made objects of study. An example is mentioned below.

It is clear, in fact, that a vast portion of the development of *new* programs is carried out by programmers on the basis of their prior experience in similar situations. Programming consists largely of choosing appropriate known techniques and adapting them to the problem at hand. That ordinary programming language is insufficient to *express* these techniques has been widely suggested by researchers interested in automating programming and program understanding. Our hypothesis (shared by others) is that *derivation structure* is the appropriate vehicle for expression; unlike programming language, derivation structure provides a way of making explicit the *rationale* for program structure.

Programming language designers have long sought to provide language constructs that reflect as closely as possible our thinking about the structure of algorithms. For example, some years ago Dijkstra and Knuth showed that nearly all uses of **goto**'s in programs were actually parts of higher control abstractions such as **while** loops, case analysis, and so on [Dijkstra71, Knuth74]. The number of distinct control constructs turned out to be small enough that they could be—and were—included as primitive in programming languages, even though they brought no additional real expressive power. The point here is that *program derivations* allow us to express our thinking about the correctness and modification of programs in a much more natural—and useful—way than do conventional proofs.

In this regard we mention the "Programmer's Apprentice" automatic programming system designed by Rich, Shrobe, and Waters at MIT [Rich78, Waters81]. One of the key notions in this system is the programming *cliché*, which is a programming-language syntactic manifestation of a program derivation pattern. It was found that it is not adequate to describe clichés purely in terms of program text; some external structuring must also be specified. For this purpose, the notion of "plan" was introduced. A plan is an abstraction based on program structure; it provides a much richer way of describing relationships in programs than ordinary program text. The primary limitations on this enterprise derive from the fact that plans are basically abstractions from program structure; they do not express evolution or *rationale* in any direct way. The connections with the progressive commitments to implementation are also not easy to formulate in this way. Therefore, we feel we have to re-examine completely the idea of derivation in order to have a notion that captures the right features of the programming process.

**Conceptualizing Program Derivation.** *Specifications* differ from programs in that they describe aspects or restrictions on the functionality of a desired algorithm without imposing constraints on *how* that functionality is to be achieved. That is, from the point of view of specification (by our definition), the means by which the desired functionality is obtained is not relevant. In this sense, specifications for programs are *static*; they constrain implementations by constraining the *relationship* between input and output

parameters. But even this distinction between input and output can be regarded as a temporal, implementation-based notion (as is seen in the example of PROLOG), so specifications that appear fully abstract often are not [Clocksin81]. Implementations, similarly, are not usually fully constrained either; programmers frequently leave many crucial representational decisions to their compilers, involving themselves in sticky details only when performance is exceptionally poor.

Thus, following [Bauer81, Schwartz73], we can be led to view this difference between specification and implementation simply as one of *degree*. But let us note here that the "wide-spectrum" languages that have been proposed are only a partial solution; it is sometimes—perhaps usually—necessary that semantic meanings of programming language constructs change as derivation proceeds. This, as is remarked below, is a form of *commitment*, and we feel it is a sensitive issue.

In *achieving* a specification by programming, then, many decisions have to be made about how abstractions used in a specification are to be realized in a limited language of actual or virtual machine operations. This involves representing abstract objects in the form of data or control structures or by a combination of both. For example, a *function* could become manifest as an array (in which indices are mapped to cell contents), or as a procedure calculating output values from input values, or as a list of input/output pairs that must be searched. The range of possibilities is vast, and a great variety are used in practice.

Indeed, programmers are so familiar with the many techniques for representation that they often jump directly from informal specification to realization without ever being too conscious of the act of choice made for the abstraction being realized. The choice of representation, however, can depend on many factors, and in practice trial and error is required to obtain the right structure, which makes the programmer more conscious of what he is doing. Of course, much backtracking and revision is required to obtain a workable *specification* in the first place, but let us remember that at this point in the discussion we are concerned only with the structural relationship between specification and implementation.

We thus move, in a program derivation, along the near continuum from specification to implementation by means of a sequence of representational decisions, or *commitments*—whether entirely consciously or not. We use the term "commitment" to refer to the process of introduction of structure that goes along with realizing or representing an abstraction. In this sense, commitment and abstraction, as processes, are inverse notions.

The commitments that allow passage from specification to implementation are linked together and, indeed, advantage is realized from them, by means of the *simplification* steps they permit us to make. That is, commitments introduce structure, which in turn facilitates simplification, which then suggests further commitments, and so on. In this way, the problem of implementation becomes one of selecting an appropriate sequence of commitments.

What is simplification? Any such notion must necessarily be based on some idea of *cost* or *complexity*, since otherwise the term "simplification" is meaningless! At specification time, the only cost is conceptual cost, and simplifications made to "improve" specifications

are intended to lower conceptual cost. But if we are to move towards implementation, this notion of conceptual cost must give way to the more usual (and better understood) notion of computational cost. If we could minimize conceptual cost and computational cost simultaneously, then there would be no need for this notion of program derivation at all. Practice, unfortunately, has shown this to be impossible, so we must develop structure in which movement can be made along the cost axis depending on the needs of the "reader" (human or mechanical). Program derivations are thus *directional* in this sense, but their *orientation* depends on whether conceptual cost or computational cost is being minimized. (Actually, we should be speaking of a cost *space*, whose axes include not only computational and conceptual costs, but also costs such as numerical accuracy.)

Thus, representational commitments increase conceptual cost, but they are necessary if computational cost is to be decreased and because *execution* of programs requires us to realize the abstractions of the specification in the limited, concrete terms that computers can accept and manipulate. Thus, our program derivation structure emerges: a progression of increasingly committed representations of programs leading from specification to implementation.

Let us recall, however, that a program derivation describes a structural relationship only; we can use it as a setting for discussion not only of the traditional problem of obtaining useful implementations from specifications, but of the inverse problem as well. We obtain a useful *specification* for a program by selectively ignoring implementation details—by a sequence of *abstraction* steps. In this scenario, the cost being minimized in simplification steps becomes conceptual cost. Although non-effective specifications are often the most natural ones, it is usually inappropriate (as we remarked above) to find *maximally* abstract specifications. In fact, for certain applications such as editors and operating systems the most useful specifications tend to be less abstract (and much larger) than for other applications. There is, of course, nothing inherent in program derivation structure that forces a certain level of abstraction in specifications. But as we understand more about how to write useful specifications, the distance between specification and implementation in program derivations will increase and the derivations will become correspondingly more useful.

Although it is not structurally necessary, it will simplify our discussion if we consider separately the business of deducing facts in derivations—facts about the situation at a particular point in a derivation or simply imported from outside bodies of knowledge. Of course, these *observation* steps don't affect *cost* in the same way that simplification or commitment/abstraction steps do, but they do provide a mechanism by which the business of *establishing* a precondition for some other step can be separated from the step itself.

We remark here that an essential goal in this development is to find a repertoire of program derivation steps that fairly closely reflect the way we think about program development informally [Floyd79]. For this reason, we have justified our division of program derivation steps into the categories of commitment/abstraction and simplification on purely philosophical grounds. This is the same kind of necessarily informal argument that some logicians use to argue that, say, Gentzen-style formal proofs are more *natural* than Hilbert-style formal proofs, and are therefore more suited to informal application. In this case,

however, it is still a bit early to tell if the proposed structure is exactly the right one. (Comparison with a number of informal and formal derivations in the literature suggests that while in some cases the structure we suggest is indeed directly apparent, in most it becomes apparent only when *aggregates* of steps are considered. For example derivations, see [Burstall77, Clark80, Green78, Paige82, Reif82].)

**Examples from Practice.** We have just argued that commitments and simplifications, along with observations, are the fundamental steps of program derivations. We wish now to show that, while this categorization is still perhaps a bit speculative, many conventional programming techniques fall very nicely into these classes of steps. We mention several specific forms that these steps can take on.

Perhaps the most immediate form of commitment is the representation of abstract structures such as functions (as discussed above), or sets, or graphs. Graphs, for example, can be represented as adjacency matrices, as linked physical structure, as relation subprograms, and so on. There are, however, other kinds of commitments that are made when passing from specification to implementation. Specifications describe relations, while executions of programs are inherently sequential activities. In order to develop implementations, then, we must determine an *order of computation*—not necessarily total, as for concurrent or nondeterministic computations. Once ordering commitments are made, then simplifications can be made that allow sections of code to take advantage of results computed earlier. For example, traversing the nodes of a graph after making a commitment to an order of computation that is consistent with a depth-first traversal allows for a very efficient implementation. It might be the case, however, that for certain applications this order of enumeration of nodes is not appropriate, and less efficient methods must be used.

A much simpler example concerning order of computation arises, say, in the usual iterative integer square root program. In this trivial program, the integer square root $i$ of a positive number $n$ is found by initializing $i$ to 0 and incrementing its value until $i^2 \leq n < (i+1)^2$. By making this commitment to testing values of $i$ in ascending order, the value of $(i+1)^2$ from a previous iteration can be used directly as the value of $i^2$ on the next. Further, the *next* value of $i^2$ can be computed directly from the preceding value by a simple addition. But note these algebraic simplifications are possible *only* because of the commitment to incrementing the value of $i$.

A less obvious form of commitment is commitment to *order of presentation.* To illustrate this kind of commitment, we mention briefly an example of a programming language that permits *avoidance* of commitment to order of presentation. In PROLOG, one defines *relations* and is permitted to access the relations in many ways. If, for example, a relation $R(x, y, z)$ is defined (possibly recursively), then the definition could be used in a straightforward way to test if a given triple is in the relation. But, in certain circumstances, it could also be used when given, say, particular values for $x$ and $z$ only, leaving the system to search for values of $y$ for which the relation holds. It is a specific language design decision to discourage explicit commitments on how defined relations are used. For many applications, this avoidance of commitment has a distinct cognitive advantage and can be realized fairly efficiently using a parameter binding mechanism based on unification. This notion, on the other hand, is so foreign to users of conventional languages that it rarely

appears in published specifications. The lesson, then, is that many commitments are made in specification concerning *input* variables and *output* variables that often unnecessarily constrain the range of possible implementations (in a program derivation setting).

On the other hand, some languages such as POP-2 provide a limited explicit mechanism for *introducing* such commitments. Partial application, called by Ershov *mixed computation*, is an example of such a mechanism [Beckman76, Ershov78]. Indeed, as Ershov points out, compilation is itself just the simplification that results naturally from an order-of-presentation commitment. In this case, procedure text and perhaps certain actual arguments are provided before others (*i.e.,* at *compile* time), and then the natural simplifications are made. Note that these "natural" simplifications may depend on deep outside theorems introduced as *observations*.

Language design that permits explicit commitments to be made or avoided certainly broadens the range of applicability of that language. In the program derivation framework, however, we seek language features for specifications that permit avoidance of the various kinds of commitments whenever possible. Machine language programs, in which very few such decisions are left unresolved, are at the other extreme of the spectrum.

Commitments are made not only to eliminate from specifications those abstractions not directly realized by computers, but also to permit *simplifications*. Let us consider, by way of example, the use of the *divide-and-conquer* paradigm to obtain the usual binary array search algorithm. The problem is to determine if a key $k$ appears in an array segment $A[\ell..u]$. The essential assumption is that the array elements are given in increasing order. We can then take advantage of this assumption after a commitment is made to testing a particular array element, say $A[m]$. (This is the "divide" step.) If the test fails, we want to test recursively the remainder of the array. A simplification allows us to use additional information about the outcome of the test to exclude more than just that single element $A[m]$ from consideration; indeed, the entire initial portion of the array segment from position $m$ to the beginning can be excluded if $A[m] < k$. We are then left with a considerably smaller segment for the recursive call (which means much less to "conquer").

Another form of simplification is the elimination of simple *redundancy*, and this is the basis for the so-called *dynamic programming* paradigm for algorithm design. The Cocke-Kasami-Younger parsing algorithm, for example, results from a simple recursive definition of the *derives* relation for Chomsky-Normal-Form context-free grammars. By making an appropriate order-of-computation commitment—which in this case is really only a partial commitment—and eliminating redundancy in the resulting definitions, an exponential-time algorithm is transformed into a polynomial-time algorithm.

## 3. The Rôle of Formalization.

If we are to succeed at building semantically based tools that will have any significance for programming methodology, our conceptualization of program derivation, which we have just discussed in general terms, must lead to some of the same kinds of understanding that logic brought to the perception of the structure of mathematical proof. The parallel must not be regarded as perfect, however. Formalization in logic is the means by which *proofs* are introduced as legitimate objects of mathematical attention; that is, a formal proof itself

becomes a mathematical structure that can be reasoned about. Our conceptualization of program derivations, similarly, must lead us to a point where program derivations are considered as *formal* structure representing the evolutions of programs. As we remarked, we are still far from being able to propose the exact details of the required formalization; but even the suggestion that it is *possible* to find such a definition raises many questions—and generally raises blood pressure as well!—since the significance of formalization in mathematics remains hotly debated.

**Who deals with formal structures?** We do not mean to suggest that programmers must deal directly with the minute formal structure of a derivation—say, the filling in of the names of the principles employed at each step. Indeed, it will be asked: do we need to formalize program derivations at all? Mathematicians do not really build formal proofs in practice; why should programmers? Real programming, like the proving of real theorems, is a process unhampered by the observance of niggling little rules, a process requiring, rather, insight and creativity. How can formalization help? Perhaps formal logic improved our understanding of the *structure* of mathematical proof, but it hasn't helped us prove *new theorems*, has it? Well then, neither will formalization help us find new programs.

The tenor of this objection is common, and it sounds very reasonable at face value. We find a basic fallacy here, however. In a way, formalization plays an even more important rôle in computer science than in mathematics because—as is obvious to all programmers—programming languages are by necessity formal languages. The programming process is a process of building and reasoning about formal program structures; people can consume informal proofs (indeed more easily than their formal counterparts), but computers do not run "informal" programs. To run, programs must be syntactically absolutely correct. Much of the good advice of program methodology is aimed at keeping the complexity of the formal programs under control, and much systems building is devoted to constructing aids for providing a programming environment allowing the programmer to rise far above the counting of parentheses and the filling in of semicolons. That is not all there is to formalization, of course.

To be fair to classical mathematics, its level of precision has improved by leaps and bounds since the turn of the century. The field of algebraic geometry is an outstanding example, and this improvement has almost nothing to do with the history of mathematical logic and logical formalization. To cite a trivial example of precision (that has been around for a very long time but required clear thinking when it was originated), consider proving, say, the convergence of some infinite series. Many manipulations must be done, and every student has experienced the problem of getting some signs wrong. Going back over the formulae until the missing minus sign is located is just exactly the same as debugging a program. Much of the language of algebra, calculus, and more advanced analysis *is* a formal language, and people prove things by learning the rules of the formalization. Today, the MACSYMA system is able to deal with these manipulations in a pretty direct way. And, much, much more of mathematics has been formalized in this sense: take large portions of algebraic topology, for example, which require heavy formal machinery now often expressed in category theory. Mathematicians are able to prove important theorems with the aid of these formalizations that were unthinkable 100 years ago. The point is that the difficulty

of the abstractions has forced *real* mathematicians to introduce what can only be called formal methods. The argument that mathematicians have with logicians, on the other hand, is over the further question of whether there is any sense in looking at a *complete* formalization of a whole proof. Often there is not.

Returning to the domain of programming, we think that there is a difference in scale and a difference in kind between programming and mathematics. It will be agreed that much of programming deals with systems: languages, compilers, interfaces. All these topics are formal by nature, as structured messages have to be interpreted and either executed or turned into something else. That is, computation is a symbol manipulation activity, and so formalization lies at the basis of all automated systems. This is the difference in kind.

But, even relatively elementary programs tend to be more complicated than elementary theorems. Why? Because in a certain sense more of their structure has to be made explicit. In mathematical literature, it is often sufficient to do one precise calculation, and for the other cases say "similarly". A proof is often more a genteel tourist guide than instructions for a treasure hunt. Programs, on the other hand, not only operate on very highly structured data, but they must do so in unobvious ways in order to be efficient. All this pressure requires a high degree of formal treatment. But, just as before, we ask: even if programming is more concerned with formalization than mathematics, must the *whole* process be formalized? This leads us directly to the next question.

**How to go: formal or informal?** Like proofs, program derivations can certainly be presented both formally and informally. Formal derivations, like any formal proof or other structure, are not intended for everyday consumption. Much misunderstanding has resulted when this important distinction has not been recognized. For example, David Gries has this pessimistic quotation on program transformation in his new book [Gries81] (p. 235).

> It is extremely difficult to understand a program by studying a long sequence of transformations; one quickly becomes lost in details or bored with the process.

We do not intend here to ascribe this misunderstanding to Gries. We suspect, rather, that his remark is a natural response to the way in which transformation sequences have sometimes been *presented* in the literature, not to their *structure*. It is just as inappropriate—to understanding—to use a sequence of *formal* transformation steps to describe a program as it is to justify a theorem by giving a full formal proof. We are quite comfortable with informal mathematical proofs; we must learn better, however, how to present transformation sequences (which are really program derivations) in an informal way so that they will be a useful tool for explaining programs.

It should not be forgotten that we have *never* had useful informal ways of justifying the correctness *and* structure of programs until recently! Indeed, most of our experience in this area is with *formal* proofs; the historical development of programming methodology has been completely unlike that of ordinary mathematics in this regard. The point of most of the early program-transformation papers, for example, was to expose new program-transformation techniques, not to give accounts of algorithms [Broy81, Burstall77, Manna79, Paige82, Wand80]. More recently, papers have appeared in which the transformational method is used to explicate existing algorithms—and occasionally even *new*

algorithms—but we have still not yet developed concise informal language for describing the transformation sequences [Clark80, Green78, Reif82].

Gries went so far as to suggest that programs subject to transformation should be proved afresh at every stage of development. We suggest, contrariwise, that the transformation process, properly presented, constitutes a much more useful proof than the usual sort of static program proof. Of course, transformations can transform proof along with program, so this requirement could always be satisfied trivially—but this is not the point.

Here is the key design problem: we must conceptualize program derivation in such a way that both informal and formal presentations will be (appropriately) useful! That is, formal program derivations will be useful only if the structure they make manifest is in some way a reflection of the way we think (intuitively) about the evolution of programs. We must build formal structures that will permit not only the *presentation* of arguments (in the form of program derivations) but also the *development* of new arguments and the adaptation and modification of old ones—by means of inferential programming techniques.

An alternative approach to formalizing our reasoning about the evolution of programs is based, rather directly, on the evolution of formal proofs in mathematics, combined with rules for generating an algorithm from the proof [Bates79, Bates82, Goad82, Kriesel82, Martin-Löf79]. Improvements to the algorithm are made either by restructuring the formal proof or through the use of automatic optimization tools. Commitment steps, in this approach, correspond to *proof-development* steps. The inferential programming approach, on the other hand, is based on the thesis that the passage from informal to formal can bring us directly into a program-oriented language, even if the "programs" are nothing more than abstract—even noneffective—specifications.

When we think again about who deals with formal structures, the answer that we hope will emerge is that the programmer deals with them on one level (just as mathematicians have to) while manipulating them in informal ways in his head or in his documentation, and the system deals with them on quite another level interacting with the programmer(s). The kind of system we would like to see built will keep track of all the little syntactical niceties as a matter of course, but at a different stage of interaction will dovetail steps of the *derivation* under direction of the programmer. This view gives us a chance to suggest a new answer to the next question.

**At what point do we find verification?** It is commonly argued that formalization inhibits the social process of acceptance of proofs. Indeed, why should we *believe* formal proofs—even if they are produced by a machine—since they are almost immune to the usual kind of social process of having friends, enemies, referees, students, and others check the details in their heads or with pencil and paper. In objecting specifically to the suggestions for methods of (automatic) program verification, De Millo, Lipton, and Perlis say at the start of their well-known paper [DeMillo79] (p. 271):

> We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem—and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail. We can't see how it's going to be able to affect anyone's confidence about programs.

Further on they are even more outspoken and say (p. 275):

> The proof by itself is nothing; only when it has been subjected to the social processes of the mathematical community does it become believable.

In this regard—though they mention the early muddy history of the "solutions" to the Four-color Conjecture—De Millo, Lipton and Perlis do not discuss the subsequent machine-aided proof of this world-famous conjecture. (Their paper was received by the editors in October 1978 and probably written much earlier. The first announcement by Haken, Appel, and Koch of the solution of the conjecture was made in September 1976, and the paper was published in September 1977, see [Haken77].) As this example fits exactly into the area of methods of proof that they are criticizing, we would like to go more into the circumstances of the proof and its believability.

Fortunately, the computer proof—a checking of a large number of cases—of the Four-color Theorem (hereafter, 4CT) has caused considerable comment, so we shall not have to be too explicit here about mathematical details and can refer to the published literature. Everyone has surely heard the statement of the problem: Does every finite planar map require only four colors to color all regions so that no two adjacent regions have the same color? The conjecture received many inadequate proofs over the years. In February 1979, in an article *The four-color problem and its philosophical significance*, Thomas Tymoczko published a long broadside [Tymoczko79] against the claimed solution in which he asserted (p. 58):

> What reason is there for saying that the 4CT is not really a theorem or that mathematicians have not really produced a proof of it? Just this: no mathematician has seen a proof of the 4CT, nor has any seen a proof that it has a proof. Moreover, it is very unlikely that any mathematician will ever see a proof of the 4CT.
>
> What reason is there, then to accept the 4CT as proved? Mathematicians know that it has a proof according to the most rigorous standards of formal proof—a computer told them! Modern high-speed computers were used to verify some crucial steps in an otherwise mathematically acceptable argument for the 4CT, and other computers were used to verify the work of the first.
>
> Thus, the answer to whether the 4CT has been proved turns on an account of the role of computers in mathematics. Even the most natural account leads to serious philosophical problems. According to that account, such use of computers in mathematics, as in the 4CT, introduces empirical experiments into mathematics. Whether or not we choose to regard the 4CT as proved, we must admit that the current proof is no traditional proof, no *a priori* deduction of a statement from premises. It is a traditional proof with a lacuna, or gap, which is filled by the results of a well-thought-out experiment. This makes the 4CT the first mathematical proposition to be known *a posteriori* and raises again for the philosophy the problem of distinguishing mathematics from natural sciences.

Subsequent commentators are far from agreeing with Tymoczko that "we are committed to changing the sense of 'theorem', or more to the point, to changing the sense of the underlying concept of 'proof.'" In the same journal in December 1980 in a reply, *Computer proof*, Paul Teller argues cogently against all Tymoczko's conclusions [Teller80]. In the very same number of the journal, Michael Detlefsen and Mark Luker in another long reply, *The four-color theorem and mathematical proof*, point out that computer checking of certain proof steps is hardly a novelty (they give several telling references and quotations) and they discuss rather fully the question of 'empiricism' [Detlefsen80]. They say (p. 804):

> We do not disagree with Tymoczko's claim that evidence of an empirical sort is utilized in the proof of the 4CT. What we find unacceptable is the claim that this is in any sense novel.

It is best for the reader to read the original paper and the replies himself to judge the issues. They make interesting reading.

Another author, however, who touches on the points closest to our present discussion is E. R. Swart in yet another reply to Tymoczko in November 1980 in an article *The philosophical implications of the four-color theorem* [Swart80]. Swart—owing to his detailed and professional knowledge of the field—points out that in fact Tymoczko fails to recognize the actual weak point of the Haken-Appel proof. Moreover, he suggests a cure. He also points out that other machine-aided proofs of the 4CT are now available, referring to extensive work of F. Allaire. He says (p. 698):

> ... Allaire's proof also involves a discharging/reducibility approach but only requires some 50 hours of computer time. It is moreover, based on an entirely different discharging procedure and a completely independently developed reducibility testing program. At the very least Allaire's proof must rank as an independent corroboration of the truth of the four-color conjecture, and there can be little doubt that even if the Haken/Appel proof is flawed the theorem is nevertheless true.

Swart goes on convincingly to support the thesis that computers are more reliable than humans in checking details and says (p. 700):

> Human beings get tired, and their attention wanders, and they are all too prone to slips of various kinds: a hand-checked proof may justifiably be said to involve a "complex set of empirical factors." Computers do not get tired and almost never introduce errors into a valid implementation of a logically impeccable algorithm.

He understands, of course, that the original algorithm design is the important point here.

In our view what Swart demonstrates in a fully documented way is just how computer-aided proof *can* be subjected to the "social process"—which is just what De Millo, Lipton and Perlis denied was going to happen. In the case of the 4CT, Swart criticizes the original method, he points out how it can be strengthened, he discusses an alternative approach, and he comments on the general reliability of algorithms. This is more attention to particulars than many results get, and in any case he has gone into the mathematical details elsewhere and clearly knows what he has been talking about. A particularly encouraging statement about the "social" nature of the activity occurs on p. 704 of his paper:

> It is perhaps appropriate to begin to draw this section to a close with some discussion of the obvious first requirement of mathematical proofs—namely, that they should be convincing. At this juncture in history the 4CT has not been properly integrated into graph theory as a whole and stands to some extent as a monument on its own, but there is little doubt that this is not its permanent lot.
>
> Indeed, it already has strong connections with at least some branches of graph theory that have no direct reliance on computer programs. Several mathematicians, such as Walter Stromquist, Frank Bernhart, and Frank Allaire, who did research on the question of reducibility also developed a coherent theory of irreducibility that is in complete agreement with the reducibility results that have been obtained thus far on the computer. Moreover, in the light of such irreducibility theory, it became possible to determine anti-configurations for all planar configurations that are not freely reducible. And Frank Allaire was able to make excellent use of such anti-configurations in finding reducers for intractable reducible configurations. Such developments can surely only serve to strengthen the confidence that mathematicians have in the truth of the 4CT.
>
> And in the years to come, when the theorem is even more inextricably intertwined with graph theory as a whole, it will seem not a little quaint to even suggest that it is not

...n *a priori* theorem with a surveyable proof. The four-color conjecture served as an excellent stimulus to graph-theoretic research, and the 4CT may continue to exert a benign influence on graph theory until such time as it has been brought into "the body of the kirk."

We think this answers De Millo, Lipton, and Perlis pretty fully. To be fair to them, they did not have very good paradigms in mind: at the time that they wrote their paper the suggestions for computer-aided verification were low-level. But Swart describes very well what happens to this kind of work when there is actual content involved. He shows that, since people are involved in formulating the problems and in evaluating the results, *thoughts* tend to occur to them. No one is going to be satisfied with the answer "VERIFIED"—they will want to know how and why and what the context is. Though the 4CT is a highly specialized problem, the algorithms developed have wider significance beyond the proof of one theorem. This is the typical direction of problem solving into generalization, as everyone knows.

Our contention is that the difficulty with the question of program verification—which has almost become a dirty word—is that the question was asked at the wrong level of abstraction. Our approach is to replace this question by the aim of correct program *development* with correctness being checked at each stage. This puts the emphasis on verification in a different light. De Millo, Lipton, and Perlis say (p. 279):

> The concept of verifiable software has been with us too long to be easily displaced. For the practice of programming, however, verifiability must not be allowed to overshadow reliability. Scientists should not confuse mathematical models with reality—and verification is nothing but a model of believability. Verifiability is not and cannot be a dominating concern in software design. Economics, deadlines, cost-benefit ratios, personal and group style, the limits of acceptable error—all these carry immensely much more weight in design than verifiability or nonverifiability.
>
> So far, there has been little philosophical discussion of making software reliable rather than verifiable. If verification adherents could redefine their efforts and reorient themselves to this goal, or if another view of software could arise that would draw on social processes of mathematics and the modest expectations of engineering, the interests of real-life programming and theoretical computer science might both be better served.

We agree with them that reliability is the key driving force and that verifiability at some time *after* a giant program has been completely written by different hands is virtually impossible. But we cannot let this be an excuse for not producing *provably* correct software, even if "correct" only means that documentation is generated concerning the *extent* to which a program is reliable. We ourselves would not be satisfied with such a weak notion of correctness, but informative documentation would be a step forward. The emphasis in our minds is on "proof"—meaning that the right *observations* from allowable ones are made at the right points in the derivation.

Once we accept the proposition that individual program derivation steps *preserve* correctness, then it is implicit in derivation structure that implementations are consistent with specifications. The confidence of users arises from their knowledge of *how* the implementation is linked to the specification, and no formal proofs need ever change hands. That is, confidence is based on the assured *existence* of the proof and not on its content or structure. It is exactly this kind of reasoning that justifies our confidence that the object code produced by programming language compilers is faithful to the source code. Once we

accept the correctness of a compiler—an acceptance which is almost always effected by a social process—we necessarily accept the correctness of the results; this is what compiler correctness *means*. In practice, we rarely bother even to inspect the results of compilations. Our treatment of transformations is going to be like our treatment of compilers—and indeed like arbitrary programs—for, once we accept their correctness, then we necessarily accept the correctness of their results.

**What are the proper analogies?** De Millo, Lipton, and Perlis contrast two analogies in their paper (p. 275):

**The Verifiers' Original Analogy**

| *Mathematics* | | *Programming* |
|---|---|---|
| theorem | ... | program |
| proof | ... | verification |

**The De Millo-Lipton-Perlis Analogy**

| *Mathematics* | | *Programming* |
|---|---|---|
| theorem | ... | specification |
| proof | ... | program |
| imaginary formal demonstration | ... | verification |

We just do not agree with this picture of the activity. We would like to revise the analogy in the light of all the foregoing discussion, since we feel that, if left in the above form, it does a disservice to the concept of formalization and its correct rôle.

**The Revised Analogy**

| *Mathematics* | | *Programming* |
|---|---|---|
| problem | ... | specification |
| theorem | ... | program |
| proof | ... | program derivation |

All of this has to be taken with a big grain of salt, since all these words can mean many things. We prefer to put "problem" in parallel with "specification," because the statement of a (mathematical) problem formulates a goal without saying how to solve the problem. Neither does a specification determine an algorithm—nor need a specification be satisfiable at all. In both cases the answer must be found, and it need not be unique.

Now theorems come in many flavors. If De Millo, Lipton, and Perlis understand by "theorem" a statement like "The problem has a solution," then we agree there is not much to choose between problem and theorem—and their analogy can stand. But many more

theorems are stated "The solution to the problem is given by this formula: ... ," and this is much more like a program. There is no hint in the statement of the theorem *why* the formula gives the answer; one must look to the proof. As we have remarked before, programs tend to be pretty explicit, so let us understand by "theorem" something more like "theorem *cum* construction" which is parallel to "program for a given specification." But if we agree that the lines of the analogy table are not meant to be read in isolation, then we can take it that each line follows on from the previous one and that our suggestions can remain in the more abbreviated form.

The word "proof," admittedly, is much harder to pin down; proofs, too, come in all flavors, and they are all too often half baked. A good proof should contain some discussion to let the poor reader know how the solution to the problem was arrived at. There are lots of tedious steps that have to be checked, but the author of the proof should have supplied some organization to the way the steps have been assembled. We are back at our conflict between formal and informal methods. De Millo, Lipton, and Perlis say (p. 275):

> There is a fundamental logical objection to verification, an objection on its own ground of formalistic rigor. Since the requirement for a program is informal and the program is formal, there must be a transition, and the transition itself must be necessarily informal. We have been distressed to learn that this proposition, which seems self-evident to us, is controversial. So we should emphasize that as antiformalists, we would not object to verification on these grounds; we only wonder how this inherently informal step fits into the formalist view. Have the adherents of verification lost sight of the informal origins of the formal objects they deal with? Is it their assertion that their formalizations are somehow incontrovertible? We must confess our confusion and dismay.

Confession is always regarded as good for the soul, but somehow we cannot accept that the authors are too sincere in this passage. They must have felt that they had dealt a death blow to the project of verification, and they did not want to gloat too much. But our whole argument in this paper is that the place for verification is *within a program derivation.* Formalization is much concerned with the *way* the steps of the derivation fit together; while the informal understanding of how the solution to the original problem is emerging is in the *choice* of the sequence of steps. (Granted, an informal reader of a program derivation may need some comments to help him remember where he is and which subgoal is in need of attention, but a mathematical proof needs some commentary also—if one is ever going to learn anything beyond an ability to recite the proof verbatim.) There is plenty of room here for the interplay between formal and informal strategy without relegating formal methods to the madhouse.

It is no argument that some mathematicians give vague proofs—so vague that one often wonders how they solve their problems at all. There are excellent authors that craft proofs that can be checked even by undergraduates. There are also proofs that have thorny steps that require machine-aided checks. As we learn to take advantage of the power of the computer, there will be many more of these proofs. Students will soon learn to use the necessary machines (perhaps before their teachers), and proofs will become more complex. There is no conflict between the formal and the informal in this way of telling the story, however, as each has its place.

In making these analogies, one must take care to assess goals. There are problems that must be faced by the computer scientist that mathematicians might regard as unutterably

boring. As far as the latter is concerned the problem is solved, and the choice of a particular computation method or a particular representation of the data is of no concern. The programmer will have to slog through many thickets in which the mathematician can see no game. The subjects are different, the interests are different, and the aims are different, but the two hunters after solutions of problems can learn from each other. Just as logic has something to tell us about proofs, we feel inferential programming can provide a framework for presenting and justifying the structure of algorithms and programs.

## 4. The Inferential Programming Problem.

The success of the inferential programming approach to program development will depend on the design and implementation of the supporting mechanical systems. There is no denying that constructing an adequate design that can be made to work in practice is a truly challenging problem! A major difficulty in building such a system is that there are many, many ways in which we can deal with derivations: extending them in one direction or another, reasoning about them, using them as a basis for constructing new derivations, and so on.

It is not our intention here to champion any particular philosophy of practical program development, nor do we intend for inferential programming systems to embody any preconceived approach to programming methodology or management. We believe, in fact, that programming, like other areas of creative endeavor, should not be too heavily shackled by form or method. *Tools* or *vehicles* for programming, therefore, must be constructed in such a way that programmers will still feel the freedom to explore in unfettered fashion. They must be given, however, the added confidence that the paths they explore are all safe ones—leading to correct implementations and faithful specifications. Awareness, in addition, has to be maintained of the various possible directions still open, and advice must be forthcoming when "navigation" decisions are to be made. The journeys may still involve experimentation and backtracking and, rarely, unprotected forays, but the experiences ultimately gained have to be captured in the form of "maps" that others can use to stay safely on course when in similar situations. Thus, as experience accumulates, programmers will find themselves more often in known—or at least easily charted—territory in which they can defer decisions to their guides, interfering only when necessity (or curiosity) demands.

It must be emphasized here that we do not regard programming as an intellectually shallow activity that can be automated simply by finding the right set of "tricks." No one in the near future will succeed in fully automating the programming process, and we must not waste our efforts in such an attempt. Instead, we feel we should focus on building powerful *interactive tools*. As our understanding of the process of programming improves, it is true that more aspects of it will be subject to complete systematization and automation—but the completely mechanical programmer is a will-o-the-wisp. For this reason, we are concentrating our investigations on finding the sorts of components a powerful interactive inferential programming system should have, and on understanding how they could aid in both the formal and heuristic aspects of programming.

**Integrating deduction.** There must be, first of all, a mechanism for carrying out the fundamental program-derivation steps. In its most primitive form, this mechanism

will apply syntactic transformations in order to make or remove commitments, to carry out simplifications, and to make straightforward observations. More advanced capabilities would include tools for manipulating and reasoning about entire derivations. All of these symbol-manipulation activities can be regarded as forms of deduction, and a powerful single general-purpose deductive mechanism such as that already used in LCF [Gordon79] could go a long way towards realizing them.

Whatever the *deductive facility* is, it must at the very least have sufficient capability that users are not bothered with having to make trivial algebraic simplifications and transformations. It goes without saying that the more difficult simplification and observation steps may require considerable activity in formal reasoning with many heuristic decisions—and perhaps with interaction from the user. Although we feel able to address the heuristic issues in a meaningful way, much thought is still required before the right style of interaction between user and system can be arrived at. More will be said on these points below, but it should be clear that the heart of an inferential programming system will be its deductive mechanism.

It seems to be a valid point to make that an effective approach to theorem proving is first to start with a suitably implemented proof-checker technology and then to add heuristic features. The success of LCF is largely due to this correct philosophical attitude. In LCF a clear distinction is made between metalanguage and object language, permitting users to focus separately on facts and the strategies that control the process of inference. A really general programming language (for LCF it is called ML) for controlling inference was also used in the AI languages PLANNER and CONNIVER. The novelty of LCF is the use of the ML type mechanism to maintain important distinctions in the object language, such as between theorems and other formulas. This permits users to experiment with proof strategies and be confident of not disturbing the underlying logic.

As the formal reasoning mechanism will be operating primarily on *program-derivation structure*, we have to ask what the actual shape of this structure will be. Since our project is not yet at the stage of implementation, we can only anticipate now the kinds of problems that will probably arise when we set out to formalize the informal understanding of derivations discussed earlier. At a first approximation, a program derivation is likely to be a directed graph in which nodes are programs and arcs are fundamental program derivation steps (*i.e.*, commitment and simplification). The simplest sort of program derivation yields a linear graph; more complex structure emerges when alternative commitments are pursued and different implementations of the same specification (or multiple specifications for the same implementation) are obtained.

Program derivations are themselves objects, and it is often easiest to obtain a *new* program derivation by a transformation on an existing derivation. A mechanism of this sort (and a language for expressing *relationships* among derivations) will have to be developed if, for example, users are to be able to create new derivations by analogy with existing ones. This question also leads us to thinking about derivation strategies and heuristics and their realizations as higher-level derivations. Much experimentation remains to be done, especially in obtaining a feeling of how deduction is to be combined with the more prosaic steps of program construction.

**Adapting conventional tools.** Many groups are currently at work building and using program-development tools. We cannot survey the whole vast field here, but we do wish to discuss some useful aspects of present efforts in order to be able to explain how the additional features we hope to add to a system will qualify it for use in what we call inferential programming. The existing programming aids help programmers operate on the *syntax* of their programs, but they generally do little to help when the *effects* of programs must be considered. It is an essential part of our thesis that powerful *semantically based* programming tools will utterly change the way in which programs are created and modified. It is not simply a matter of adding these new features to our standard tools but, rather, of creating an entirely new attitude towards the programming process.

Consider *structured editors*, for example. With these tools, programmers are allowed to explore alternative syntactic constructions as they manipulate the text of their programs. They are thus freed from concerns relating to the *syntactical* correctness of their programs. We assert, similarly, that programming tools that operate on derivation structure will help programmers explore a range of implementations while essentially freeing them from *semantical* correctness concerns. The change in attitude here makes programming move closer to problem solving.

The first "automatic programming" tools were of course the *compilers.* Incremental compilers and associated programming environments—as seen, for example, in modern LISP systems—are among our most powerful contemporary programming tools. Perhaps the principal reason the LISP environment is so attractive is that users have tremendous freedom to modify the text of programs *and* sample their executions without having to follow the rigid discipline of edit/compile/test associated with traditional compilers. In such an environment, users can respond to problems by investigating the execution context of the problem, then, perhaps, by making a local change (while still in the context), and finally by continuing execution. Immediate response and adaptation to small problems is possible, and radical context switches are not required except in unusual cases. An inferential programming environment, similarly, should not force general retreat when small problems develop. Rather, it must allow an *incremental* approach to the manipulation of program derivations, which is again a change of attitude.

Very little success has been experienced in applying formal techniques from programming logics to reasoning about *very large programs*, and indeed a very natural worry about the inferential programming paradigm is its ability to scale up. Will inferential programming systems ever be sold other than in the toy shops? The answer lies partly in the development and use of powerful *modularization* techniques for both programs *and* program derivations. Modularization is an important concern of systems builders, and such a facility for inferential programming would allow individual parts of a large program to be derived independently and then combined together in such a way that all possible interactions can be anticipated. Here is a case where the right attitudes are already familiar, except they have not been applied sufficiently to deduction.

By anticipating interactions, *version control* becomes a much more precise activity—another very critical concern in large-scale programming. Current systems for version control, as seen in GANDALF and MasterScope, must apply a necessarily conservative strategy

to the task. Because they are unable to make inferences about the *effects* of changes, *any* change must be treated as a *major* change, and analysis and perhaps recompilation is necessary for those modules that might *possibly* be affected. If a semantic component were added to these systems, then changes would need to be propagated only to those modules that were truly affected. We feel this would encourage more extensive experimentation.

*Type checking*, both at compile time (*static* checking) and at run time (*dynamic* checking), is among the essential mechanisms that programming languages have provided to help programmers protect lines of *abstraction* in their programs. The trend towards *self-documenting programs* has brought a variety of new kinds of abstraction facilities in programming language designs, along with correspondingly complex languages of types and type-inference algorithms. In program derivations, the lines of abstraction are drawn *between* programs rather than *within* them, so the need for complex typing mechanisms may diminish appreciably. But, we will still have to devote considerable effort to the design of the typing mechanisms used in the *formal* language in which the program derivations themselves are expressed. Like formal proofs—and indeed any formal objects (even programs)—program derivations have many, many internal consistency requirements, and a suitably rich typing mechanism can make the process of checking and maintaining these consistency requirements largely mechanical. Programmers who want to study and use strongly typed languages can still be accomodated, in any case, even if we feel we can shift part of the burden of type checking in program development to other phases of the process.

**Developing heuristics.** An understanding of *meaning* does not necessarily imply a command of *technique*. Students can develop a reasonably deep understanding of the foundations of calculus without developing any skill at solving integrals. Similarly, many students are quite adept at integration, but have little understanding of the fundamentals, so we must conclude conversely that a command of technique does not imply a deep understanding of meaning. Though these remarks are truisms, they suggest that if we are to design a useful deductive facility, we must provide methods for introducing not only new knowledge to the database, but also information regarding how the facts are to be *used*.

Inferential programming tools will become more applicable as the heuristic knowledge they embody increases. Programmers will be tinkering constantly with their personal stores of heuristic knowledge in order to make them more powerful and flexible. It is necessary, however, to protect the database of *facts* from this constant tinkering, so our deductive mechanisms must be designed to keep correctness issues separate from the heuristic mechanism.

As we remarked in Section 2, the bulk of programming activity—whether in the modification of existing programs or in the creation of new programs—is carried out, often consciously, by analogy with past experience. Analogical inference is a fundamentally heuristic activity, involving search and pattern recognition. A system that supports it must store representations of past experience, aid programmers in finding useful analogies with derivation patterns in the store, help them select the most fruitful analogies, and finally allow them to adapt the store of knowledge as needs and understanding change [Carbonell82]. As our grasp of this kind of heuristic reasoning improves, our tools will

become better at helping programmers find not only new analogies but also new *kinds* of analogies. The heuristic mechanism of an inferential programming system must facilitate this kind of reasoning.

A simple example will illustrate the sort of reasoning that might go on. Consider the derivation of a program for numbering the nodes of a tree in preorder. A specification, say for generalized tree traversal, is committed to visit tree nodes in preorder. From this preorder enumeration algorithm, an algorithm for explicit preorder *numbering* is then derived. Now, the most natural way to derive the *postorder* numbering program is to follow this derivation, but with a slightly different commitment. These derivations are closely related because, although the commitments are different, the pattern of simplification steps is essentially the same. That is, at some level of abstraction, the same simplification activity is being performed. This could be the case even if the new program structure that results may not have any obvious resemblance with the original program.

Roughly, an analogy exists between two phenomena if there is a "close" general phenomenon that captures essential qualities of both. If we are to reason effectively by analogy, then we will need to develop a language for program derivations that has an abstraction mechanism that is rich enough to *express* these generalizations. Thus, we must not only introduce conceptualizations concerning the fundamental program derivation steps, but about common *patterns* of their usage. Present programming languages do not, in general, have sufficiently rich abstraction mechanisms even to express directly the various kinds of analogies that can exist among *programs*. Were these analogies expressible, they still would not be nearly as useful (and would not reflect our intuitive thinking nearly as closely) as the sorts of analogies that exist among *derivations*. The issue boils down to this: Can we find program derivation *abstractions* that can capture the common patterns of programming activity?

## 5. Programs of the Future.

Just as twenty years ago we learned to move away from the details of object code by thinking about control and data structures more abstractly, we are learning now to move away from the details of algorithm, representation, and implementation by thinking instead about the qualities we desire of them and how they might be chosen. Thus, rather than leading to programs we can no longer understand, the use of inferential programming techniques will lead to a different view of how programs are to be presented.

Stripped down to essentials, our claim is that the "programs" of the future will in fact be descriptions of program derivations. Documentation methods based on stepwise-refinement methodologies are already strong evidence that there is movement toward this approach. These documentation methods also provide support for the hypothesis that program derivations offer a more intuitive and revealing way of *explaining* programs than do conventional proofs of correctness. The conventional proofs may succeed in convincing the reader of the correctness of an algorithm without giving him any hint of *why* the algorithm works or how it came about. On the other hand, a derivation may be thought of as an especially well-structured "constructive" proof of correctness of the algorithm,

taking the reader step by step from an initial abstract algorithm he accepts as meeting the specifications of the problem to a highly connected and efficient implementation of it.

We shall not arrive at inferential programming overnight, however, because the very act of producing a complete derivation requires a programmer to express some of his previously unexpressed intuitions. Thus, it may often be harder to produce a complete program derivation than simply to write code for an implementation. The additional effort is justified by the fact that the explicit representation of the derivation sequence facilitates analysis, proof, and, most importantly, eventual modification of the programs derived. Many tools remain to be built to make this kind of programming possible. We believe, nevertheless, that the first comprehensive steps are becoming feasible, and we hope, further, that the arguments we have put forward in this paper will make the outcome seem worth the effort.

## Bibliography


[Balzer81] Balzer, R., *Transformational implementation: an example.* IEEE Transactions on Software Engineering, Vol. SE-7, No. 1, pp. 3–14, 1981.

[Barstow80] Barstow, D. R., *The roles of knowledge and deduction in algorithm design.* Yale Research Report 178, April 1980.

[Bates79] Bates, J. L., *A logic for correct program development.* Ph.D. Thesis, Cornell University, 1979.

[Bates82] Bates, J. L. and R. L. Constable, *Proofs as programs.* Cornell University Technical Report, 1982.

[Bauer81] Bauer, F. L., *et al., Programming in a wide spectrum language: a collection of examples.* Science of Computer Programming, Vol. 1, pp. 73–114, 1981.

[Bauer82] Bauer, F. L., *From specifications to machine code: program construction through formal reasoning.* Sixth International Conference on Software Engineering, 1982.

[Beckman76] Beckman, L., A. Haraldsson, Ö. Oskarsson, and E. Sandewall, *A partial evaluator and its use as a programming tool.* Artificial Intelligence, Vol. 7, pp. 319–357, 1976.

[Broy81] Broy, M. and P. Pepper, *Program development as a formal activity.* IEEE Transactions on Software Engineering, Vol. SE-7, No. 1, pp. 14–22, 1981.

[Burstall77] Burstall, R. M. and J. Darlington, *A transformation system for developing recursive programs.* Journal of the ACM, Vol. 24, No. 1, pp. 44–67, 1977.

[Carbonell82] Carbonell, J., *Learning by analogy: formulating and generalizing plans from past experience.* Carnegie-Mellon University Technical Report, 1982.

[Cheatham72] Cheatham, T. E. and B. Wegbreit, *A laboratory for the study of automatic programming.* AFIPS Spring Joint Computer Conference, Vol. 40, 1972.

[Cheatham79] Cheatham, T. E., J. A. Townley, and G. H. Holloway, *A system for program refinement.* Fourth International Conference on Software Engineering, pp. 53–63, 1979.

[Clark80] Clark, K. and J. Darlington, *Algorithm classification through synthesis.* Computer Journal, Vol. 23, No. 1, 1980.

[Clocksin81] Clocksin, W. F. and C. S. Mellish, **Programming in PROLOG.** Springer-Verlag, 1981.

[DeMillo79] De Millo, R. A., R. J. Lipton, and A. J. Perlis, *Social processes and proofs of theorems and programs.* Communications of the ACM, Vol. 22, No. 5, pp. 271–280, 1979.

[Detlefsen80] Detlefsen, M. and M. Luker, *The four-color theorem and mathematical proof.* The Journal of Philosophy, Vol. 77, No. 12, pp. 803–820, 1980.

[Dijkstra71] Dijkstra, E. W., *Notes on structured programming.* In: **Structured Programming.** (O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Eds.) Academic Press, 1971.

[Ershov78] Ershov, A. P., *On the essence of compilation.* Formal Descriptions of Programming Concepts, E. J. Neuhold, ed., North-Holland, 1978.

[Feather82] Feather, M. S., *A system for assisting program transformation.* ACM Transactions on Programming Languages and Systems, Vol. 4, No. 1, pp. 1–20, 1982.

[Floyd79] Floyd, R. W., *The paradigms of programming.* Communications of the ACM, Vol. 22, No. 8, pp. 455–460, 1979.

[Goad82] Goad, C., *Automatic construction of special-purpose programs.* 6th Conference on Automated Deduction, 1982.

[Gordon79] Gordon, M. J., Milner, A. J., and C. P. Wadsworth, **Edinburgh LCF.** Springer-Verlag Lecture Notes in Computer Science, 1979.

[Green78] Green C. C. and D. R. Barstow, *On program synthesis knowledge.* Artificial Intelligence, Vol. 10, p. 241, 1978.

[Green81] Green, C., *et al., Research on knowledge-based programming and algorithm design.* Kestrel Institute Technical Report, 1981.

[Gries81] Gries, D., **The science of computer programming.** Springer-Verlag, 1981.

[Haken77] Haken, W., K. Appel, and J. Koch, *Every planar map is four-colorable.* Illinois Journal of Mathematics, Vol. 21, No. 84, pp. 429–567, 1977.

[Knuth74] Knuth, D. E., *Structured programming with goto statements.* Computing Surveys, Vol. 6, No. 4, pp. 261–301, 1974.

[Kriesel81] Kriesel, G., *Neglected possibilities of processing assertions and proofs mechanically: choice of problems and data.* In: **University-Level Computer-Assisted Instruction at Stanford: 1968–1980.** Stanford University, 1981.

[Manna79] Manna Z. and R. Waldinger, *Synthesis: dreams ⇒ programs.* IEEE Transactions on Software Engineering, Vol. SE-5, No. 4, July 1979.

[Martin-Löf79] Martin-Löf, P., *Constructive mathematics and computer programming.* 6th International Congress for Logic, Methodology and Philosophy of Science, 1979.

[Paige82] Paige, R. and S. Koenig, *Finite differencing of computable expressions.* ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, pp. 402–454, 1982.

[Reif82] Reif, J. and W. L. Scherlis, *Deriving efficient graph algorithms.* Carnegie-Mellon University Technical Report, 1982.

[Rich78] Rich, C. and H. Shrobe, *Initial report on a Lisp programmer's apprentice.* IEEE Transactions on Software Engineering, Vol. SE-4, No. 6, pp. 456–467, 1978.

[Scherlis81] Scherlis, W. L., *Program improvement by internal specialization.* Eighth Symposium on Principles of Programming Languages, pp. 41–49, 1981.

[Schwartz73] Schwartz, J. T., *On programming, an interim report on the SETL project.* Courant Institute of Mathematical Sciences, New York University, 1973.

[Schwartz77] Schwartz, J. T., *On correct program technology.* Courant Institute of Mathematical Sciences, New York University, 1977.

[Swart80] Swart, E. R., *The philosophical implications of the four-color problem.* American Mathematical Monthly, Vol. 87, No. 9, pp. 697–707, 1980.

[Swartout82] Swartout, W. and R. Balzer, *On the inevitable intertwining of specification and implementation.* Communications of the ACM, Vol. 25, No. 7, pp. 438–440, 1982.

[Teller80] Teller, P., *Computer proof.* The Journal of Philosophy, Vol. 77, No. 12, pp. 803–820, 1980.

[Tymoczko79] Tymoczko, T., *The four-color problem and its philosophical significance.* The Journal of Philosophy, Vol. 66, No. 2, pp. 57–83, 1979.

[Wand80] Wand M., *Continuation-based program transformation strategies.* Journal of the ACM, Vol. 27, No. 1, pp. 164–180, 1980.

[Waters81] Waters, R. C., *A knowledge based program editor.* Seventh International Joint Conference on Artificial Intelligence, Vancouver, 1981.

[Wile81] Wile, D. S., *Program developments as formal objects.* USC/Information Sciences Institute Technical Report, 1981.

[Wirth71] Wirth, N., *Program development by stepwise refinement.* Communications of the ACM, Vol. 14, No. 4, pp. 221–227, 1971.