1. Introduction

The introduction decribes the goals of this research, defends these goals against criticism that has been made of program verification, summarizes the results of the paper, and gives some examples that illustrate the value of verification in compilation, the kinds of theorems that a verifier must prove mechanically, and the notation that will be used in the rest of the paper.

1. Goals

Programming systems that include mechanical verification as part of the compilation process will make programs more efficient, reliable, and flexible—if they can ever be made practical. The main reason such systems have not yet progressed beyond the experimental stage is that they depend on mechanical theorem provers, which have been too slow and unreliable. The goal of the research described in this paper is the construction of a mechanical theorem prover that is fast and reliable for the class of problems encountered in program verification.

If program verification fulfills its greatest potential, future programmers will write programs that are as reliable as their specifications, and write specifications in so clear a language that they are self-evidently correct. However, research in the last decade has revealed formidable obstacles in the path to this goal, and whether it can ever be reached is a subject of debate. Many people have concluded that verification methods have no immediate practical value in typical software projects, but this conclusion is probably wrong: There is immediate potential for verification methods in establishing invariants that, while not strong enough to show the total correctness of the program, are still of great practical value in compilation.

An *invariant* is a statement that is invariantly true at some point during program execution, for example "whenever the procedure **qsort** is entered, l will be less than r," or "if the **if** statement at line 1405 succeeds, then $rlink(p) \neq$ nil," or "when the GOSUB returns, i will be between 1 and n." A program verifier's business is to verify the invariants suggested to it by the programmer and compiler, and perhaps to ferret some out by itself in the process.

If appropriate invariants have been established, a compiler can safely dispense with runtime checks on array selections and pointer dereferences, can safely produce code that returns records to the free list instead of waiting for a garbage collector to do so, can simplify polymorphic or otherwise generalized module bodies into the appropriate instance required by the program at hand, and can generally provide greater reliability and flexibility without sacrificing the speed of the compiled program.

It is illuminating to view these compilation-related invariants as generalized type declarations. Conventional type declarations can be viewed as invariants; for example, to declare that p has the type **procedure**(a : **array**[1, 10] **of integer**) is (more or less) to declare that at entry to the body of p, it is invariantly true that a is an array of ten integers. A compiler for a conventional hard-typed language checks that procedure and module interfaces respect the type system; in other words the compiler verifies that the declared invariants really are invariants. The restrictions and complications in existing type systems are introduced to facilitate this verification, and could be removed if more sophisticated verification methods were used. For example, suppose we are writing a procedure p whose argument is an array of integers, $a_1, \ldots, a_n$, and that we do not know n in advance, but will instead assume the existence of a 0th element a_0 that contains the number n.

Using a programming system based on a mechanical theorem-prover, we could label the entry of the procedure with the invariant:

$$\mathbf{array}(a) \wedge \mathbf{lob}(a) = 0 \wedge \mathbf{hib}(a) \geq 0 \wedge \mathbf{hib}(a) = a(0)$$

where **array** is a predicate that is true of arrays, and **lob**(a) and **hib**(a) are the low and high bounds of the array a, respectively. A verifying compiler would refuse to compile a program until it had verified that the entry condition was satisfied at every call to p, and that only valid components of a were accessed in the body of p. For realistic programs, such a verification requires only simple reasoning about inequalities, using automatic techniques that are well-understood and efficient.

Of course, one could add to Pascal or Algol a particular type to handle this problem, though some care must be taken to prevent the program from changing $a(0)$. We might call the type "arbitrarily long but not dynamically variable array of integers." In this way many programming languages have acquired an elaborate collection of types, whose rules of use are intricate and error-prone. PL/I, for example, uses several different types for string variables. The invariants verified by a mechanical theorem prover can provide more precise specifications for procedures and modules, using a type structure that is simple, powerful, and general.

Thus the invariants of a program range from the trivial ("at line 6, the type of x is **integer**") to the very difficult ("at line 32767, the output file is a valid compiled image of the input file"). Existing compilers check the trivial ones; experimental verifiers grapple with the most difficult ones; between these extremes there is a class of invariants, rich by comparison to conventional type systems, that could be managed by a practical verifying compiler. Section 4 contains a sketch of such a type system, and some example programs that use it.

The basic verification problem is to determine whether a program P meets given entry and exit specifications F and G, in the sense that if F is true when P is started, G is invariantly true when P finishes. If P is a large program, it will be composed of smaller pieces, each of which will be specified with its own entry and exit invariants. P will be verified under the assumption that the pieces work correctly, and the pieces will be verified separately. By repeating this process, the general verification problem is reduced to the problem of verifying that some elementary construct of the programming language meets given entry or exit conditions. If the programming language has an appropriate semantic defintion, it will be possible to answer such questions directly (assuming a mechanical theorem prover is available). For example, it might be a semantic rule that $P(x)$ is true at exit from $x \leftarrow t$ if and only if $P(t)$ is true at entry. Such rules for relating predicates before and after execution of some programming language construct are called *predicate transformation semantics* for the construct. Program verification, at least as it is now known, cannot be performed on languages that lack predicate transformation semantics.

Most languages were designed with no consideration for verifiability, and as a consequence it is infeasible to give them predicate transformation semantics. (Their state transformations depend on so many obscure properties of the state that the corresponding predicate transformations are unmanagable.) To obtain the benefits of program verification, it is necessary to plan for them during language design. This is a controversial subject, since many people fear that languages with suitably regular semantics will be unequal to the tasks of real-world programming, or in any case unsuited to the tastes of most programmers. On the opposing side, a number of languages have been proposed, or outlined, with very regular semantics (see for example references [19, 36]) that are (arguably) adequate for general-purpose programming. This paper, while primarily concerned with mechanical theorem-proving, continues this line of research by using a new language for its examples. As a certain amount of discussion of the semantics of the language used for examples is inevitable in any case, outlining a new language is not too much of a digression.

In summary, the main goal of this research is to develop and refine mechanical theorem-proving techniques, and a secondary goal is to outline a programming language and verifying compiler that take advantage of the techniques.

2. Defense of program verification

There are mountainous obstacles to program verification, and many people say flatly, "it will never work." Before summarizing the results of this paper in section 3, let us consider several objections that are frequently raised against program verification, some of them spurious, others serious.

The first objection that we will consider is based on theoretical results. Many logical theories can be shown to be undecidable, or to be decidable only by algorithms with super-exponential time bounds. These results are sometimes construed as obstacles to mechanical theorem-proving, but to do so is to underestimate their scope: such lower bounds apply not only to machine proofs, but to any proofs. For example, [23] proves that there are true statements of length n about geometry whose shortest proofs have length $2^{\Omega(n)}$; thus there is a theorem of geometry that can be squeezed on a page, although any of its proofs would more than fill the universe with fine print. Whether we use pencil and paper or electronic computers, such theorems are secure forever from our attack; but this is irrelevant to the question of whether mechanical theorem provers can find the proofs that mathematicians find. It is also demonstrably difficult to find a winning move in an arbitrary position in the game of checkers (in the precise sense that the problem is P-space complete on an n by n board), but, as anybody who has seen the construction in the proof [25] will agree, it would be mistaken to infer from this that computers cannot play checkers.

The second objection is based on the difficulty of specifying a program. To verify mechanically that a program has some property, the property must be defined in a formal language. Thus the important question arises: can the formal specification of the desired properties of a program be any simpler than the program itself? For example, the APL statement $x \leftarrow \iota n$, which sets x to the value $(1, 2, \ldots, n)$, is as simple as any formal specification for it; and at the other extreme, the formal specification of all the properties that are desired of some operating system, compiler or editor might be as formidable as the listing of a program that has the properties. These examples show that it is naïve to talk without qualification about verifying the "correctness" of a program; the best that can be done is to verify that one complicated object (the program) is "consistent," in some complicated technical sense, with another complicated object (the set of specifications). Thus—the objection goes—verification does not lead to reliability, but instead multiplies the possibilities for error.

One answer to this question is that adding redundancy makes it more likely that errors will show up as inconsistencies. A more forceful answer is that the result of a computation can be more clearly specified in a language designed for such specifications than in a language designed for describing computations themselves, because specification languages are not required to be efficiently executable (or even executable at all). Many programming languages sacrifice some efficiency for power, clarity, and ease of expression; but even languages (like APL) that go a long way in this direction do not approach the concise and powerful notation common in mathematical discourse. If all concern for efficiency is abandoned, complicated programs become simpler; if in addition the language they are written in is designed for clarity only, they become simpler still.

It is a surprisingly difficult task to design a specification language, and those of most verifiers are rather cumbersome. Designers of programming languages may think that all their problems would vanish if efficiency were no longer required; but it is still a challenge to find a harmonious language that is expressive but not baroque. The expressiveness of the specification language is a critical factor in program verification; perhaps this has not received the emphasis that it deserves.

The third objection is a practical one; it goes something like this: Since everyone knows that errors are omnipresent, the attempt to verify a program will either never succeed, or succeed only erroneously. There is a lot of truth in this objection. There may be errors in the specifications, in the verifier, in the operating system, in the hardware; substitute zero for oh in a few key places and a bank's new verified accounting system will set everyone's balance to zero. Foolish people advertise program verification along these lines: "Think what peace of mind you will have when the verifier finally says '`verified`,' and you can relax in the mathematical certainty that no errors remain."

The answer to the objection is that the purpose of verification is not to produce peace of mind, but to find bugs in programs. A verifier would indeed be useless if failed verifications did not point the way to bugs in the program; but in fact they do. The value of a verifier is in no way contingent on its being bug-free. As it is prudent to assume that there are bugs in everything, the message at the successful exit of program verifiers should be changed from "`Verified`" to "`Sorry, can't find any more errors.`"

In fact, "bug-catcher" is a better name than "verifier" for the kind of system aimed at in this paper. Most compilers provide a crude kind of bug-catching in that they check that the usage of a variable is consistent with its type (and to achieve this they use an unpleasantly rigid type system). Section 4 describes a more expressive type system that can be handled with the aid of a verifier, allowing the system to catch many more errors at compile time than a traditional compiler could ever hope to.

The last objection that we will consider is the main one raised by Richard De Millo, Richard Lipton, and Alan Perlis in their article [18]. They summarize their argument as follows: "The aim of program verification, an attempt to make programming more mathematics-like, is to increase dramatically one's confidence in the correct functioning of a piece of software, and the device that verifiers use to achieve this goal is a long chain of formal, deductive logic. 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. ... 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."

Briefly, the article argues that program verifications are to programming as "imaginary formal demonstrations" are to mathematics: they have no rôle in practice. The authors quote Poincaré: "... if it requires twenty-seven equations to establish that 1 is a number, how many will it require to demonstrate a real theorem?" They write "The *Principia Mathematica* was the crowning achievement of the formalists. It was also the deathblow for the formalist view. ... Russell failed, in three enormous, taxing volumes, to get beyond the elementary facts of arithmetic." Formal methods (the authors declare) are important only in theory; in practice mathematicians use informal methods suitable for human social interaction.

It is true that we have more confidence in long-standing theorems than in new results, and that the social process accounts for this. But instead of focusing on a theorem after it is published, or in its formative stages when it may be the subject of excited talk and blackboard discussions, consider the crucial stage during which the theorem is transformed from an intuitive idea, perhaps supported by diagrams or vague scribbles, into the quasi-formal language that is required by scientific journals. This is usually done alone, since it is hard to get anyone to read your error-filled drafts; many mistakes are found in this stage (in fact some results simply evaporate); and it is in this stage that a mathematican, alone with a thick sheaf of definitions and formulas, relies most heavily on formal methods. Since this stage is the one most nearly analogous to the coding of a program, we can expect formal methods to be valuable in coding.

In other words, this objection is based on a questionable view of the rôle of formal methods in mathematics and computer programming. Since the whole thrust of this paper is towards formal methods, we will consider the objection carefully. Let us consider three problems, one from analysis, one from topology, and one from computer programming:

- Determine the area bounded by the x-axis, by the line $x = 1$, and by the parabola $y = x^2$.
- Show that a simple closed curve divides the plane into two regions.
- Show that if two circularly-linked lists of records are "spliced" by exchanging any link in one list with any link in the other list, the result is a circular list containing all records of the original two lists.

The first two problems are much more managable than the third, because of the availability of appropriate formal methods. Thus there are standard formal definitions for the notions of area and of connected region, but it is not obvious how to state the third problem formally. The solutions to the first two problems can also be presented formally: for the first, we have

$$\int_0^1 x^2\,dx = \frac{x^3}{3}\Big|_0^1 = 1/3;$$

for the second problem, let $\beta(G)$ be the betti number of the group G, f an injection of S^1 into S^2, and $|f|$ the range of f. Then the number of components of the complement of $|f|$ is

$$\beta(H_0(S^2 - |f|)) = 1 + \beta(\tilde{H}_0(S^2 - |f|)) = 1 + \beta(H^1(S^1)) = 1 + \beta(Z) = 2.$$

By contrast, the third problem requires drawing a picture, which is no proof at all, or an inductive proof that runs several pages. Surely this is not because of the inherent difficulty of the problem, but rather because appropriate formal methods have not been developed. It is instructive to remember how difficult the first two problems are to solve from first principles; before the development of symbolic algebra and calculus, Archimedes gave a long geometric construction for the "quadrature of the parabola"; before the development of algebraic topology, the Jordan Theorem was proved by elementary methods, but the proof is notoriously long and tricky.

These examples show that a question like "is the Jordan Theorem within the range of a mechanical theorem-prover" is nonsense. It depends on what formal methods the machine is armed with. Perlis *et al.* assume that a mechanical theorem-prover must start with primitive notions and construct a "foundational" formal proof, in a language like set theory; but this is not true. Such foundational demonstrations are relevant only when formal symbolic methods are themselves the objects of study, as in Hilbert's program or Whitehead and Russell's work. But outside of the foundations of mathematics, formal symbolic methods are used in practical reasoning, as the above examples showed. MACSYMA is an example of a program which does formal manipulation in a practical formal domain.

The case for formal methods is summarized by Hilbert [27], pp. 441–2: "It is an error to believe that rigor in the proof is the enemy of simplicity. On the contrary we find it confirmed by numerous examples that the rigorous method is at the same time the simpler and the more easily comprehended. The very effort for rigor forces us to find out simpler methods for proof. It also frequently leads the way to methods which are more capable of development than the old methods of less rigor. ... I think that wherever, from the side of the theory of knowledge or in geometry, or from the theories of natural or physical sciences, mathematical ideas come up, the problem arises for mathematical science to investigate the principles underlying these ideas and so to establish them upon a simple and complete system of axioms, that the exactness of the new ideas and their applicability to deduction shall be in no respect inferior to those of the old arithmetical concepts."

It is a tough challenge to build a system in which we can reason about programs with the exactness and deductive ease with which we reason about arithmetic. The results in this paper are offered as a step toward this goal.

3. Summary of results

Most of the results in this paper are algorithms for mechanical theorem-proving in the conventional first-order predicate calculus with equality. These results are presented first, since they do not depend on any non-standard logical systems. For reasons described later in this section, the predicate calculus is not expressive enough to describe programs that manipulate linked data structures. Therefore, in section 16 we extend the predicate calculus with recursive function definitions. We defer the semantic definition of the programming language used in the examples until section 17, since its semantics are related to those of recursive functions, and because this masses the formal technical parts of the paper toward the end where they are out of the casual reader's way. Thus, some of the notations used in the first sections of the paper precede their formal definitions; we give informal definitions for them in section 4. The last few sections of the paper discuss the mechanical theorem proving problem in the presence of recursive function definitions.

The algorithms for theorem-proving in the conventional predicate calculus, described in sections 5 to 14, are essentially those used by the Stanford Pascal Verifier [56] to reason about conjunctions of literals. This part of the Verifier's theorem prover was implemented by Derek Oppen and the author. Essentially, the theorem-prover's data stucture is shared by several cooperating "satisfiability procedures." A *satisfiability procedure* for a logical theory is a program that solves the "satisfiability problem" for the theory. The *satisfiability problem* for a theory is the problem of determining the satisfiability of a conjunction of literals (signed atomic formulas) from the theory.

The satisfiability problem is to be distinguished from the *decision problem*, in which quantified statements are allowed. The problem of proving an arbitrary quantifier-free formula F in a theory can be reduced to the satisfiability problem; for example, $A \wedge B \supset C$ is valid if and only if $A \wedge B \wedge \neg C$ is unsatisfiable. If the boolean structure of F is complicated, this reduction may produce exponentially many instances of the satisfiability problem. If we excude such formulas (since no practical way to handle them is known), then the complexity of the satisfiability problem for a theory is a good measure of how difficult it is to reason mechanically in the theory.

The theorem-prover for the Stanford Pascal Verifier contains satisfiability procedures for four logical theories: the theory $\mathcal{R}$ of the real numbers under addition, the theory $\mathcal{E}$ of equality with uninterpreted function symbols, the theory $\mathcal{L}$ of Lisp list structure, and the theory $\mathcal{A}$ of arrays. These theories are defined in section 5; here we briefly describe their satisfiability problems.

The satisfiability problem for $\mathcal{R}$ is equivalent to the linear programming problem. Khachian [29] describes a polynomial-time algorithm for this problem, but the algorithm has not been tested in practice. The satisfiability procedure for $\mathcal{R}$ is based on the simplex algorithm, which takes exponential time in the worst case, but is fast in practice. An example of a conjunction that it will prove inconsistent is

$$y > 2x + 1 \wedge y > 1 - x \wedge y < 0.$$

In general, the simplex algorithm will determine the satisfiability over the reals of any conjunction of linear inequalities or equalities. Although almost all theorem provers used in program verification have special routines for arithmetic formulas, the simplex algorithm has rarely been used, possibly because most implementations of the algorithm are designed to handle large problems arising in operations research, and such implementations are not suitable for mechanical theorem proving.