Proof-Checking, Theorem-Proving, and Program Verification

Robert S. Boyer and J Strother Moore

Technical Report #35
January 1983

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

Abstract

This article consists of three parts: a tutorial introduction to a computer program that proves theorems by induction; a brief description of recent applications of that theorem-prover; and a discussion of several nontechnical aspects of the problem of building automatic theorem-provers. The theorem-prover described has proved theorems such as the uniqueness of prime factorizations, Fermat's theorem, and the recursive unsolvability of the halting problem.

The article is addressed to those who know nothing about automatic theorem-proving but would like a glimpse of one such system. This article definitely does not provide a balanced view of all automatic theorem-proving, the literature of which is already rather large and technical.[1]Good places to start on the technical literature are [Loveland 78] and [Bledsoe 77]. Nor do we describe the details of our theorem-proving system, but they can be found in the books, articles, and technical reports that we reference.

In our opinion, progress in automatic theorem-proving is largely a function of the mathematical ability of those attempting to build such systems. We encourage good mathematicians to work in the field.

The work reported here was supported in part by NSF Grant MCS-8202943 and ONR Contract N00014-81-K-0634.

[1]

1. Tutorial.

1.1. *From Proof-Checking to Fully Automatic Proof.* A *formal proof* is a finite sequence of formulas, each member of which is either an *axiom* or the result of applying a *rule of inference* to previous members of the sequence. Typical rules of inference are *modus ponens* and the substitution of equals for equals. A grammar for formulas, a collection of axioms, and a collection of rules of inference together define a logical *theory*.

For the usual theories of mathematics, e.g. set theory or number theory, it is a relatively modest exercise to write a program called a *proof checker* that will check, in a reasonable amount of time, whether a given sequence of formulas is a proof. For some theories, e.g. the propositional calculus, it is possible to write a computer program called a *decision procedure* that will determine whether any given formula has a proof in the theory. But for the usual theories of mathematics, it is theoretically impossible to write a decision procedure. On the other hand, it is theoretically possible to write a *semi-decision procedure*, that is, a program which will find a proof for any given formula if there is one, but which may run forever if there is no proof. For example, one can write the practically useless program that will systematically generate all the proofs in a theory. The challenge of automatic theorem-proving is to write computer programs that find proofs in a reasonable or practical amount of time.

One of the main techniques used to meet this challenge is the invention of *heuristic* proof techniques -- algorithms that analyze the problem at hand and pursue sound and plausible lines of reasoning. Unlike decision procedures, heuristics are not guaranteed to find a proof if one exists. Because the ordinary kinds of mathematical theories, such as number theory and set theory, are undecidable, heuristics will inevitably be a part of any completely automatic theorem-proving system.

While automatic theorem-provers have occasionally contributed to the proofs of new results in mathematics,[2] the kinds of proofs discovered by today's programs would be considered trivial by most mathematicians. But an automatic theorem-prover need not be a first rate mathematician to be useful. It would be a major accomplishment with far-reaching practical consequences to produce an automatic theorem-proving program that could follow and detect errors in mathematical proofs described at the level of graduate mathematics textbooks. We will explain this remark when we discuss the applications of our theorem-prover.

1.2. *Our Automatic Theorem-Prover.* Our work on automatic theorem-proving can be regarded as an attempt to construct a *high-level proof checker* for elementary number theory and recursive function theory. The basic axioms of our theory are those of Peano arithmetic, plus similar axioms defining ordered pairs. The Peano axioms characterize the natural numbers as follows. 0 is a natural number; if i is a natural number, so is the ‘‘successor’’ of i, usually written s(i); s(i) is never 0; s(i) is s(j) iff i is j. In addition, we are provided with the principle of mathematical induction as a rule of inference: To prove that any formula P(n) is a theorem for all natural numbers n, it is sufficient to prove P(0) and to prove that for all natural numbers i, P(i) implies P(s(i)).

Starting from such concepts the user of our system introduces such recursive definitions as ‘‘sum,’’ ‘‘product,’’ and ‘‘remainder’’ and uses the theorem-prover to prove theorems about them. Among the theorems proved by our machine are:

- the existence and uniqueness of prime factorizations [Boyer & Moore 79]
- Fermat's theorem: $M^{(p-1)} = 1$ (mod p) if p is prime and does not divide M [Boyer & Moore 84a]
- Wilson's theorem: p-1! = -1 (mod p) if p is prime [Russinoff 83]
- Gauss' law of quadratic reciprocity
- the existence of nonprimitive recursive functions
- the soundness and completeness of a propositional calculus decision procedure similar to the Wang algorithm [Boyer & Moore 79]
- the Turing completeness of the Pure LISP programming language [Boyer&Moore 84]

[2] Cf. the papers of Wos and Winker and of Chou in this volume.

- the recursive unsolvability of the halting problem [Boyer & Moore 84b]

To guide our system towards finding proofs for these theorems, the user of the system suggested intermediate steps for the proofs. But the total amount of assistance given by the user was approximately the text that one might find in a graduate level exposition of these theorems.

1.3. *Three Heuristics used by Our Program.* When, in searching for a proof, does a mathematician decide to make an argument by induction, and how does he decide on which formula ‘‘P’’ to do induction? Answers to this question are likely to involve a heuristic guess. Furthermore, there are many variants on the induction principle some of which may be more appropriate for a given problem than others, e.g., course-of-values, simultaneous induction on several variables, induction up to certain bounds. The major emphasis of our work has been the development of a heuristic for mechanizing mathematical induction.

The heuristic, which takes several pages to describe fully [Boyer & Moore 79], considers the definitions of the recursive functions that are mentioned in the conjecture and selects an induction argument that is the ‘‘dual’’ of some combination of the definitions. We illustrate our induction heuristic in the next section of this paper.

In addition, our program contains many other heuristics. The most complex select instances of axioms, definitions, and previously proved lemmas to use in a proof. For example, suppose some function symbol f is defined recursively. When does one decide to replace a ‘‘call’’ of f by the definition of f? The main heuristic our program uses is to check whether the recursive calls of f that would be introduced are already in the conjecture at hand. As will be illustrated below, hypotheses about such recursive calls are frequently provided by the induction heuristic.

A third important heuristic in our theorem-proving system is generalization, i.e. considering a harder problem than the one at hand. Generalization seems to be *necessary* in order to get certain conjectures proved inductively, but generalization is a very dangerous business. The most common form of generalization that our system uses is to ‘‘throw away’’ an induction hypothesis once it has been ‘‘used.’’

1.4. *Two Examples*. We will now illustrate our induction, expansion, and generalization heuristics by describing our system’s proofs of two theorems from elementary number theory: the associativity of multiplication and Fermat’s theorem.

The addition function may be defined recursively as follows:

$$\begin{aligned} 0+y &= y \\ s(x)+y &= s(x+y). \end{aligned}$$

Our program admits such equations as new axioms only after proving that one and only one function satisfies them.

We define multiplication in terms of addition:

$$\begin{aligned} 0 * y &= 0 \\ s(x) * y &= y+(x * y). \end{aligned}$$

Suppose the user of our program now submits to the system the conjecture: $(x * y) * z = x *(y * z)$. How does the theorem-prover proceed?

Our program decides to prove this by induction, after ruling out such possible ‘‘moves’’ as considering the cases and expanding some of the function definitions. How does it choose which induction to try? Consider for a moment a simple induction on x. Let p be the conjecture we are trying to prove. The *base case* is formed by replacing all the x’s in p by 0. The *induction step* is an implication from the *induction hypothesis* to the *induction conclusion*. The induction hypothesis is p. The induction conclusion is formed from p by replacing all the x’s by s(x). Thus, the $x * y$ in the induction hypothesis becomes $s(x) * y$ in the induction conclusion. But by the recursive definition of ‘‘$*$,’’ $s(x) * y$ is equal to $y+\underline{x *}\ \underline{y}$. We say that $x * y$ has ‘‘stepped through’’ the induction on x because, after simplification, it appears in both the induction hypothesis and the induction conclusion.

Given the recursive definition of ‘‘$*$,’’ the occurrence of $x* y$ in the conjecture we are trying to prove suggests a simple induction on x. The occurrence of the term $y* z$ in the conjecture suggests a simple induction on y, but another term in the conjecture, namely $x* y$, does not step through an induction on y. By such considerations our induction heuristic elects to induct on x.

The base case is trivial: both sides reduce to 0 by the definition of ‘‘$*$.’’ The induction step is more interesting. The hypothesis is

hyp: $(x* y)* z = x*(y* z)$

and the conclusion is

conc1: $(s(x)* y* z = s(x)*(y* z)$.

Our program attacks this problem first by simplifying the terms appearing in it. Consider the term $s(x)* y$ in the conclusion. By definition, this term is equal to $y+(x* y)$. Since $x* y$ already occurs in the conjecture at hand, namely, in the hypothesis, our program elects to replace $s(x)* y$ by $y+(x* y)$. By such expansions the program reduces the induction conclusion to

conc2: $(y+(x* y))* z = (y* z)+(x*(y* z))$.

Since further expansion of any term produces terms not already in the conjecture, our program stops expanding definitions at this point.

Next, the program tries to use its induction hypothesis, hyp. The right hand side of hyp, $(x*(y* z))$, has stepped through the induction and emerged inside the right hand side of the simplified conclusion, conc2. This permits the program to use its induction hypothesis by substituting the left hand side for the right in conc2. The result is:

conc3: $(y+(x* y))* z = (y* z)+((x* y)* z)$.

However, although its goal is to prove that hyp implies conc3, our program adopts the stronger goal of proving conc3, without any hypothesis, on the grounds that the induction hypothesis has been used and should not contaminate future goals. In addition, because the term $x* y$ also stepped through the induction and now appears on both sides of the equality, the program decides to adopt an even stronger generalization, obtained by replacing $x* y$ in conc3 by the new variable w:

conc4: $(y+w)* z = (y* z)+(w* z)$.

Observe that this sequence of heuristics has led the program to ‘‘guess’’ that multiplication distributes over addition. The program proves this by induction and further expansion. Thus, the system proves the associativity of multiplication without any guidance from the user. It takes our program about 10 seconds on a DEC 2060 in Interlisp to produce the proof described above.

Once the theorem-prover proves a lemma it remembers it for future use. For example, the associativity of multiplication would be used to reassociate any instance of $(x* y)* z$ to the corresponding instance of $x*(y* z)$. By having the theorem-prover prove key lemmas the user can lead it to the proofs of complicated theorems.

Below we exhibit the proof of Fermat’s theorem. Concepts used in the theorem and proof are introduced with recursive definitions, just as we introduced ‘‘+’’ and ‘‘$*$’’ above. Each English sentence below corresponds to one formula (lemma) typed by the user and proved by the system. Several of the lines require induction to prove. The proof below was constructed after the system had proved many of the theorems in Chapter V of Hardy and Wright’s *An Introduction to the Theory of Numbers* [Hardy and Wright 79].

Fermat’s Theorem: If p is prime and does not divide M, $M^{p-1} \bmod p = 1$.

Proof. Let S(n,M,p) be the sequence ($n* M \bmod p$, $(n-1)* M \bmod p$, ..., $1* M \bmod p$).

In the text below we adopt the convention that p is a prime that does not divide M.

The product of the elements of S(n,M,p) mod p is equal to $n!* M^n \bmod p$.

Observe that if $i<j<p$, then $j* M \bmod p$ is not a member of S(i,M,p) (Hint: induct on i). Hence, if $n<p$, then no

element of S(n,M,p) occurs twice. Furthermore, each element of S(n,M,p) is positive, each is less than or equal to p-1, and there are n elements.

Thus, from the Pigeon Hole Principle we have that the product of the elements of S(p-1,M,p) is (p-1)!. But we also have that the product of the elements of S(p-1,M,p) mod p is (p-1)!* M^{p-1} mod p. Hence, Fermat's theorem. Q.E.D.

2. Applications. In this section we describe some of the applications of our theorem-prover. To do so we must first elaborate our remark above that the production of a good ''high level proof checker'' would have far-reaching practical significance.

Computer programs may be regarded as formal mathematical objects whose correctness can be proved in exactly the sense that theorems are proved. A ''bug'' in a computer program represents either (a) the failure of the programmer to prove that the program does what it is supposed to do or (b) the failure of someone, be it the programmer or his employer, to specify clearly what the program was supposed to do. In principle, bugs of the first variety can be eliminated by requiring that program proofs be mechanically checked. Nor is this a mere theoretical possibility. Widespread research into ''program verification'' suggests that the cost of mechanically checking the proofs of programs is currently somewhere between 2 and 30 times as great as the normal development cost. To our knowledge, the largest program mechanically verified to date consists of 4,211 lines of executable high level code [Smith, et. al. 81]. The major, perhaps the only serious, difficulty in further reducing the cost is the development of better high-level proof-checkers.

There are two traditional types of program verification: Floyd-style [Floyd 67] and McCarthy-style [McCarthy 63]. Our theorem-prover is used in both types of program verification.

The Floyd-style, which has its roots in the classic Goldstine and von Neumann reports [von Neumann 61], handles the usual kind of programming language, of which FORTRAN is perhaps the best example. In this style of verification, certain points in the flowchart representation of a program are annotated with mathematical assertions about what is ''always true'' about the program variables and the input whenever ''control'' reaches such points. By exploring all possible paths from one assertion to the next and analyzing the effects of intervening program statements it is possible to reduce the correctness of the program to the problem of proving certain derived formulas called *verification conditions*. Furthermore, this reduction can be done mechanically once the program has been properly annotated with assertions. The computer program that produces the theorems to be proved from the annotated program is called a *verification condition generator*.

We have written a verification condition generator for a subset of ANSI FORTRAN 66 and 77 and we use our theorem-prover to prove the resulting verification conditions. We make the following claim about our verifier:

> If a FORTRAN subprogram is accepted and proved by our system and the program can be loaded onto a FORTRAN processor that meets the ANSI specification of FORTRAN [USAS X3.9-1966 66, Ansi X3.9-1978 78] and certain parameterized constraints on the accuracy of arithmetic, then any invocation of the program in an environment satisfying the input condition of the program will terminate without run-time errors and will produce an environment satisfying the output condition of the program.

Among the FORTRAN programs we have proved correct mechanically are a fast string searching algorithm [Boyer & Moore 81a], an integer square root algorithm based on Newton's method, and a linear time majority vote algorithm [Boyer & Moore 82]. However, merely browsing through our description of the verification condition generator for FORTRAN [Boyer & Moore 81a] -- in which we describe how to handle COMMON statements, second level definition, aliasing, undefined variables, and other arcane features -- is enough to convince most people that it is at best awkward to verify programs written in programming languages of the von Neumann style.

The McCarthy-style of program verification eschews programming languages such as FORTRAN and instead takes as the programming language a mathematical language, i.e. one in which axioms and conjectures can be stated. For example, McCarthy's language LISP [McCarthy 65] defines programs using lambda abstraction and recursion equations. A more recent language by Backus, the author of FORTRAN [Backus 78], is based upon combinators rather than lambda abstraction. The increasingly popular logic programming languages [Kowalski 79] are based on the first order predicate calculus.

It is our experience that most programs are much easier to verify if they are written in such programming languages, for several reasons:

- It is not necessary for the user of the verification system to shift constantly from one language to the another, i.e. from the programming language to the logical language.
- The tedious problems of storage allocation and deallocation are handled transparently by logical languages, but must be managed explicitly by FORTRAN style languages.

Among the McCarthy-style program verification problems that our automatic theorem-proving system has solved are:

- The correctness of a simple compiler [Boyer & Moore 79] and parser [Gloess 80]
- The soundness of an arithmetic simplifier [Boyer & Moore 81b], which is actually part of our theorem-prover
- The invertibility of the RSA public key encryption algorithm [Boyer & Moore 84a], which requires proving that if p and q are distinct primes, n is p∗ q, M<n, and e and d are multiplicative inverses in the ring of integers modulo (p-1)∗(q-1) then $(M^e \text{ mod } n)^d \text{ mod } n = M$.
- The termination, over the integers, of the Takeuchi function [Moore 79]:

```
Tak(x,y,z) = if x #≤# y then y
             else Tak(Tak(x-1, y, z),
                      Tak(y-1, z, x),
                      Tak(z-1, x, y)).
```

The later is a nontrivial theorem that we think would tax any mathematician for more than a few minutes.

Beyond these two traditional kinds of program verification, there are several new kinds of program verification that are emerging and to which our theorem-prover has been applied.

- The mechanical verification of concurrent, or parallel programs, has received much less attention than it deserves. A major reason, perhaps, is that new, improved methods for specifying and proving such programs by hand are being developed almost daily. Included here is the verification of networks, as opposed to systems resident on single computers. One mechanization of network verification has been based upon our theorem-prover [DiVito 82].
- The mechanical or even hand verification of real-time programs has been almost ignored. We have made a minor investigation [Boyer, et. al. 82] in which we use our theorem-prover to prove that a simple program keeps a vehicle ‘‘on course’’ in a varying cross wind. A major problem in real time control verification is the specification of the real world with which such programs are supposed to interact. In addition, timing and interrupt handling are major problems.
- The verification of specifications, i.e., proving properties about program specifications rather than about the programs themselves, has received a surprising amount of attention. The major property checked is a certain type of ‘‘security.’’ The federal government has issued RFQs for major systems with a requirement that the specifications be mechanically checked for security. One such checker, which uses our theorem-prover, is Feiertag’s [Feiertag 81].

3. Nontechnical Issues.

3.1. *Developing Heuristics.* Our experience with developing heuristics has convinced us of three doctrines.

First, it is easy to ‘‘wire’’ any particular proof into a theorem-prover. Changes to a theorem-prover that lead to no proofs besides the examples the author had in mind when he made the changes are to be eschewed. To help us avoid this pitfall, we do not permit ourselves to use in the code for our theorem-prover the name of any logical function except the primitives of our theory. Thus, in a certain sense our program behaves the same way whether the user names his factorial function ‘‘!’’ or ‘‘FACT.’’ Nevertheless, it is possible to cheat and build in subroutines that recognize when the user has defined certain functions, without ever mentioning them by name in the code. In the end, one is forced to evaluate an automatic theorem-prover by how good it is when applied to ‘‘new’’ problems. To this end we make it a habit not to change our theorem-prover’s heuristics to solve a new problem, but rather to solve the problem with the old version of the system (thereby getting valuable information about the current arrangement of heuristics). Once we have successfully tackled a new problem we consider how we might have changed the system to have made that problem easier.

Second, it is much easier to invent heuristics than to evaluate them. Generally, heuristics are motivated by a few examples. What is not so easy to see is the effect a candidate heuristic will have on other examples. In the development of our system, we have adopted the discipline of making approximately sure that our system can do whatever it used to be able to do when we add or improve a heuristic by the brute force, and expensive, technique of running the new system on all the old problems. With this filter, we have thrown out far more heuristics than we have retained.

Third, combining heuristics with other heuristics or decision procedures cannot be usefully accomplished by merely pasting them together. On numerous occasions, we have been asked, ‘‘Why don’t you incorporate into your system the decision procedures of so and so?’’ The answer is that adding new proof techniques is unlikely to be profitable unless the new techniques are tightly interwoven with the old. For example, we have recently added decision procedures for both ‘‘linear arithmetic’’ and ‘‘complete equality.’’ In both cases, we first tried adding ‘‘black boxes’’ to our system that contained the code for these decision procedures, with the idea that we would periodically pass the current conjecture to those boxes. We found this approach practically useless because it almost never happened that our current conjecture was merely a consequence of linear arithmetic or pure equality. Instead, we found it necessary to interweave code for the decision procedures with the already very complicated code that heuristically selects instances of previously proved theorems because equality and arithmetic reasoning are so often necessary to relieve the hypotheses of lemmas.

3.2. *System Engineering*. There is a surprisingly large amount of work to building an automatic theorem-proving system besides developing and coding the basic mathematical techniques for finding proofs. This extra work is largely due to the fact that humans will be using the system. Even if the number of serious users of the system is small (in our case it is about 15), we have found it cost-effective to devote a lot of time to the following issues.

OUTPUT. Understanding what an automatic theorem-prover has to say can be taxing, especially if heuristics are involved, because one not only wants to know what the system has done but ‘‘why’’ it has done it. Perhaps twenty percent of our system is devoted to describing what is going on and why. We have found it worth the time to make the output appear in literate English prose and good typographic style. The output routine is sufficiently complex to merit a special programming language. The code for reporting what the system does is necessarily intertwined with the code for deciding what to do. Changing the heuristics can force major changes to the output routines.

ERROR-RECOVERY. Because to err is human, we allow the user to recover from ‘‘mistakes.’’ The development of a complex proof with perhaps hundreds of lemmas seems inevitably to result in false starts. It is amazingly difficult to type perfectly accurate definitions and theorems. But implementing techniques for ‘‘backing up’’ and editing takes more work that it might seem.

STATE SAVING. The development of a large proof frequently requires many working days. It is necessary to be able to save the logical state of the system -- e.g., the axioms, definitions, and proved lemmas -- so that the work of one day can be continued another. Such a data base constitutes a ‘‘library’’ and much time can be spent designing and implementing such a library facility and (especially) building up libraries of useful lemmas.

RELIABILITY. In writing a one-off experimental automatic theorem-proving system, there is a great temptation to cut corners. For example, one can avoid checking that conjectures are well-formed formulas or he can fail to define exactly the mathematical theory in which the proofs are to be found. We have found it desirable but expensive to do our best to make our system ‘‘impenetrable.’’ Because our system has not been mechanically verified, it probably has errors. But there are no errors or holes of which we are aware. We used to offer to jump off the Golden Gate Bridge if someone found an error in our system that would cause it to ‘‘prove’’ a non-theorem. However, when the first such bug was found by Topher Cooper of Digital Equipment Corporation, we merely awarded him the first Golden Gate Bridge award and moved to Texas.[3] He is the only

[3] A subroutine for calculating the value of a primitive function on constants contained a bug that caused it to deliver the wrong answer on certain constants axiomatized by the user.

recipient to date.

TOOLS. It is perhaps an occupational hazard of researchers in artificial intelligence that they become involved in ‘‘tool building.’’ That is, instead of merely getting on with the job of writing programs, they spend a lot of time writing programs to help them write programs. We have suffered from this hazard. Among the ‘‘tools’’ implemented have been several text editors, an elaborate syntax checker for our own code that catches our common programming errors, and devices for overcoming the 1 megabyte memory address space limitation of Interlisp-10.

COMPUTERS. During the last 15 years, obtaining a decent computing environment for doing research on automatic theorem-proving has usually meant having access to an expensive machine with a large address space like a Digital Equipment Corporation 2060, costing around $1,000,000. This major problem is fortunately disappearing rapidly, due to the emergence of LISP machines sold by LMI, Symbolics, and Xerox at well under $100,000.

4. Acknowledgments. Our joint work began in 1971 in Edinburgh, Scotland, under Science Research Council support to Bernard Meltzer of the Metamathematics Unit of the University of Edinburgh. At SRI and at the University of Texas our continuing benefactors have been Thomas Keenan of NSF and Robert Grafton and Marvin Denicoff of ONR, to whom we are deeply grateful.

BIBLIOGRAPHY

[Ansi X3.9-1978 78]
American National Standards Institute, Inc.
American National Standard Programming Language FORTRAN.
Technical Report ANSI X3.9-1978, American National Standards Institute, Inc., 1430 Broadway, N.Y. 10018, April, 1978.

[Backus 78] J. Backus.
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs.
Comm. ACM 21:616-641, August, 1978.

[Bledsoe 77] W. W. Bledsoe.
Non-resolution Theorem Proving.
Artificial Intelligence 9:1-36, 1977.

[Boyer & Moore 79]
R. S. Boyer and J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Boyer & Moore 81a]
R. S. Boyer and J S. Moore.
A Verification Condition Generator for FORTRAN.
In R. S. Boyer and J S. Moore (editors), *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[Boyer & Moore 81b]
R. S. Boyer and J S. Moore.
Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures.
In R. S. Boyer and J S. Moore (editors), *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[Boyer & Moore 82]
R. S. Boyer and J S. Moore.
MJRTY - A Fast Majority Vote Algorithm.
Technical Report ICSCA-CMP-32, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982.

[Boyer & Moore 84a]
R. S. Boyer and J S. Moore.
Proof Checking the RSA Public Key Encryption Algorithm.
American Mathematical Monthly 91(3):181-189, 1984.

[Boyer & Moore 84b]
R. S. Boyer and J S. Moore.
A Mechanical Proof of the Unsolvability of the Halting Problem.
JACM 31(3):441-458, 1984.

[Boyer&Moore 84]
R. S. Boyer and J S. Moore.
A Mechanical Proof of the Turing Completeness of Pure Lisp.
In W.W. Bledsoe and D.W. Loveland (editors), *Automated Theorem Proving: After 25 Years*, pages 133-167. American Mathematical Society, Providence, R.I., 1984.

[Boyer, et. al. 82] R. S. Boyer, M. W. Green and J S. Moore.
The Use of a Formal Simulator to Verify a Simple Real Time Control Program.
Technical Report ICSA-CMP-29, University of Texas at Austin, 1982.

[DiVito 82] Benedetto Lorenzo Di Vito.
Verification of Communications Protcols and Abstract Process Models.
PhD Thesis ICSCA-CMP-25, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982.

[Feiertag 81] Richard J. Feiertag.
A Technique for Proving Specifications are Multilevel Secure.
Technical Report CSL-109, SRI International, 1981.

[Floyd 67] R. Floyd.
Assigning Meanings to Programs.
In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics*, pages 19-32. American Mathematical Society, Providence, Rhode Island, 1967.

[Gloess 80] P. Y. Gloess.
An Experiment with the Boyer-Moore Theorem Prover: A Proof of the Correctness of a Simple Parser of Expressions.
In *5th Conference on Automated Deduction, Lecture Notes in Computer Science*, pages 154-169. Springer Verlag, 1980.

[Hardy and Wright 79]
G. H. Hardy and E. M. Wright.
An Introduction to the Theory of Numbers.
Oxford University Press, 1979.

[Kowalski 79] R. Kowalski.
Logic for Problem Solving.
Elsevier North Holland, Inc., New York, 1979.

[Loveland 78] D. Loveland.
Automated Theorem Proving: A Logical Basis.
North Holland, Amsterdam, 1978.

[McCarthy 63] J. McCarthy.
A Basis for a Mathematical Theory of Computation.
In P. Braffort and D. Hershberg (editors), *Computer Programming and Formal Systems*. North-Holland Publishing Company, Amsterdam, The Netherlands, 1963.

[McCarthy 65] J. McCarthy, et al.
LISP 1.5 Programmer's Manual.
The MIT Press, Cambridge, Massachusetts, 1965.

[Moore 79] J S. Moore.
A Mechanical Proof of the Termination of Takeuchi’s Function.
Information Processing Letters 9(4):176-181, 1979.

[Russinoff 83] David M. Russinoff.
A Mechanical Proof of Wilson’s Theorem.
Masters Thesis, Department of Computer Sciences, University of Texas at Austin, 1983.

[Smith, et. al. 81] M. Smith, A. Siebert, B. DiVitto, and D. Good.
A Verified Encrypted Packet Interface.
SIGSOFT 6(3), 1981.

[USAS X3.9-1966 66]
United States of America Standards Institute.
USA Standard FORTRAN.
Technical Report USAS X3.9-1966, United States of America Standards Institute, 10 East 40th Street, New York, New York 10016, 1966.

[von Neumann 61]
J. von Neumann.
John von Neumann, Collected Works, Volume V.
Pergamon Press, Oxford, 1961.