A Theorem Prover for a Computational Logic

Robert S. Boyer[1]
J Strother Moore

Technical Report 54
April, 1990

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This report was the keynote address at the 10th International Conference on Automated Deduction, July, 1990 in Kaiserslautern, W. Germany.

[1]Mailing address: Computer Sciences Department, University of Texas at Austin, Austin, TX 78712, USA

Abstract

We briefly review a mechanical theorem-prover for a logic of recursive functions over finitely generated objects including the integers, ordered pairs, and symbols. The prover, known both as NQTHM and as the Boyer-Moore prover, contains a mechanized principle of induction and implementations of linear resolution, rewriting, and arithmetic decision procedures. We describe some applications of the prover, including a proof of the correct implementation of a higher level language on a microprocessor defined at the gate level. We also describe the ongoing project of recoding the entire prover as an applicative function within its own logic.

1 Introduction

We feel honored to be invited to give the keynote address for CADE-10. We thank Mark Stickel and the program committee for the invitation.

It has been suggested that we discuss our theorem prover and its application to proving the correctness of computations. We have been working on our prover, on and off, since about 1972 [9]. This prover is known both as the *Boyer-Moore theorem prover* and as *NQTHM*. (pronounced *en-que-thum*, an acronym for ‘‘New, Quantified THeoreM Prover,’’ an uninspired parochialism that has taken on a life of its own). The details of our prover and its applications have been extensively presented in several books and articles. In fact, from these publications the prover has been recoded by at least three other groups. In this paper, we will (a) very briefly review the prover and its applications, (b) provide pointers to the literature on the prover and its applications, and (c) discuss ACL2, a new development of the prover which involves recoding it in its own logic, a subset of applicative Common Lisp.

In the subsequent discussion, we will make reference to two books, which are the main references on NQTHM. They are (a) *A Computational Logic* [11] which we will abbreviate as ‘‘ACL’’ and (b) *A Computational Logic Handbook* [18] which we will abbreviate as ‘‘ACLH’’. Although a decade old, ACL still provides a rather accurate description of many of the prover’s heuristics and some simple applications, whereas the much more recent ACLH accurately describes the current logic and user interface.

2 The Logic

Although many theorem provers, especially those of the resolution tradition, are designed to work with arbitrary collections of first order axioms, NQTHM is designed to be used mainly with the fixed set of axioms we provide, typically augmented by a number of definitions provided by the NQTHM user. Questions one might ask about the NQTHM theory are ‘‘What are the well-formed formulas, what are the axioms, and what are the rules of inference?’’ The precise answers to these questions may be found in Chapter 4 of ACLH. Roughly speaking, in that chapter, we present our logic (the Boyer-Moore Logic or NQTHM Logic, as it is sometimes known) by starting from standard first order logic as in [70] and then adding some axioms that describe certain data structures, including the integers, ordered pairs, and symbols. We include in the logic a principle of definition for recursive functions over these data structures. Among our rules of inference is a schema for proof by induction. This schema would be merely a derived rule of inference were we to cast our induction axioms in the traditional form.

The syntax of our logic is close to that of Lisp. In fact, from the time we started writing our prover [9] we have regarded it as a theorem prover for a theory of Lisp functions. Some of the earliest theorems we proved mechanically were inspired by some of McCarthy’s seminal papers on the logic of Lisp, including [54], [56], and [58]. Because Lisp may be viewed as both a logic and a programming language, we have always found it a most natural setting in which to express theorems about computations and other parts of constructive mathematics.

Both the axioms of the NQTHM logic and the conjectures it entertains are quantifier free, or, more precisely, implicitly universally quantified ‘‘on the far outside.’’ In fact, NQTHM does not include rules for manipulating quantifiers at all. However, by using recursive functions, we are able to express many of the things that one usually expresses with quantifiers when dealing with ‘‘finite’’ objects such as trees of integers. For example, to state and prove the uniqueness and existence of prime factorizations [11], we define recursive functions which factor integers and which compute whether two finite sequences of integers are permutations of one another. This practice of using recursive functions to do work one might do with quantifiers may have been originated by Skolem in [71], who was perhaps the earliest to demonstrate that arithmetic could be built up using entirely constructive methods. Skolem’s program is further carried out in [34].

Here is an example of a definition that one might give to the prover for a function that appends, i.e. concatenates, two lists:

```
Definition.
(APP X Y)
   =
(IF (LISTP X)
    (CONS (CAR X) (APP (CDR X) Y))
    Y)
```

In rough English, this definition says that to append a list **X** to a list **Y**, if **X** is nonemtpy, then construct (i.e. **CONS**) the list whose first element is the first element of **X**, i.e. **(CAR X)** , and whose other elements are the result of appending the rest of **X**, i.e. **(CDR X)**, and **Y**. On the other hand, if **X** is empty, just return **Y**.

A simple example of a theorem that one might ask NQTHM to prove is the associativity of **APP**, which one would state as

```
Theorem.
(APP (APP X Y) Z) = (APP X (APP Y Z))
```

We discuss a proof of this theorem below when we describe the induction heuristic.

3 The Prover

3.1 How to Get a Copy

NQTHM is a Common Lisp program whose source files require about one million characters. NQTHM runs in a variety of Common Lisps including Lucid, Allegro, Symbolics, and KCL. It is publicly available, but a license is now required. We have recently started to require a license to keep track of copies at the strong suggestion of one of our sponsors. We previously distributed the same code without copyright or license. At the time of this writing, a copy may be obtained without fee by anonymous ftp from Internet site cli.com (start with the file /pub/nqthm/README) or on tape for a modest fee by writing to the authors at Computational Logic, 1717 W. 6th St., Austin, Texas 78703. The currently released version of NQTHM was first released in July of 1988, and no bugs affecting soundness have been reported as of the time of this writing. The chapter of ACLH on installation describes in complete detail how to bring up NQTHM from the sources.

3.2 WARNING: Difficulty of Use

It is hard, perhaps impossible, to use NQTHM effectively without investing a substantial amount of time learning how to use it. To avoid disappointment, a prospective user should probably be prepared first to understand most of two rather long books, ACL and ACLH. Almost all of the successful users of NQTHM have in fact also taken a course from us at the University of Texas at Austin on proving theorems in our logic. Based upon teaching related courses at Stanford, our former student N. Shankar advises that a user unfamiliar with the heuristics employed in the prover, as described in great detail in ACL, is very unlikely to direct the prover to prove anything significant. Besides precisely describing the logic of NQTHM, ACLH also serves as a user's manual, describing in great detail all of the commands with which one can direct the prover.

3.3 Heuristic Character of NQTHM

NQTHM is a *heuristic* theorem-prover. By heuristic, we mean that we have coded guessing strategies for searching through the space of possible proofs for conjectures. For example, NQTHM guesses when it is best to cut off ‘‘back chaining’’. If the guess is wrong, which it can easily be, then no proof may be found. As another example, NQTHM often guesses an induction to try, when all other proof techniques cease to be applicable. If the guess is wrong, then NQTHM will irrevocably go chasing down a search path that is probably totally fruitless. On the other hand, because NQTHM does have heuristics, NQTHM is able to find proofs for what we believe is a remarkable number of theorems. One crude measure of the effectiveness of NQTHM is that it is always is able to make an above average grade on the final examinations we give to our students in an introductory graduate course on proving theorems in the NQTHM logic.

We were inspired in part to build a theorem prover that is heuristic by the success of W. W. Bledsoe [4], [5] in writing such theorem provers, including one that guessed inductions based upon the terms in the conjecture. One of the major concerns in the literature on automated reasoning in the 60's and 70's was with the completeness of proof procedures. NQTHM is certainly not complete, except when guided by a knowledgeable user.

3.4 Induction

Perhaps the most important heuristic in NQTHM is the induction heuristic. The key to the success of our induction heuristic is that it is closely tied to the principle of recursive definition which we employ. For example, to prove the associativity of **APP**, mentioned above, NQTHM would guess to induction on **X** by **CDR**, i.e. by the length of the first argument. This induction mirrors the way that **APP** recurses. An important part of the induction heuristic is filtering out suggested inductions which are ‘‘not likely’’ to work, such as the induction on **Y** by **CDR** in the theorem above.

The NQTHM practice of not permitting quantification but of permitting the user to define recursive functions to express what might otherwise require quantification has an effect of forcing the user to hint implicitly how to prove conjectures: try inductions that mirror the definitions of the recursive functions used in the conjectures. This heuristic is startlingly successful for the NQTHM logic.

3.5 Simplification

Besides induction, the other most important theorem proving component in NQTHM is the simplifier. The simplifier combines rewriting (cf. [74] and [4]) with linear and binary resolution and subsumption (cf. [65] and [53]). The simplifier also includes a semi-decision procedure for a part of arithmetic, based upon ideas in [36].

An aspect of our simplifier that accounts for much of its effectiveness is a ‘‘type set’’ facility which keeps track, for each expression actively under consideration, a bit-mask’s worth of information indicating a conservative estimate of the ‘‘type’’ of the expression in terms of the basic data types of the NQTHM Logic. The type set procedure, like most of the current simplifier, is described in ACL.

3.6 Other Heuristics

Of considerably less significance than induction or simplification are various NQTHM routines which are named ‘‘elimination of destructors,’’ ‘‘cross fertilization,’’ ‘‘generalization,’’and ‘‘elimination of irrelevance.’’ The structure of NQTHM, at least as it was in 1979, is described in complete detail in ACL. That work remains a largely accurate description of NQTHM except for (a) the integration of the arithmetic decision procedure, described in [19], (b) the addition of metafunctions and an efficient representation for large constant terms, described in [12], (c) some simplifications of the induction machinery that have never been documented, (d) the axiomatization of an interpreter for partial recursive functions [20], and (e) an implementation of a derived rule of inference called ‘‘functional instantiation’’ [8].

4 The Importance of the User in Finding Proofs

Although NQTHM is quite capable of finding proofs for some simple theorems with which even graduate students may struggle, we think of NQTHM as more of a proof checker than as a theorem-prover. What do we mean by this distinction? It is perhaps not possible to spell out clearly what the distinction is. However, whenever we have in mind an interesting theorem for NQTHM to prove, we almost always expect to have to suggest to NQTHM what the main intermediate steps to the proof are. We do expect NQTHM to do a great deal of tedious work filling in minor details. And when filling in such minor details, NQTHM very often exposes minor oversights in our statement of theorems. The situation is entirely different for some ‘‘real theorem-provers,’’ such as those of Wu and Chou [27], which one expects to decide quickly any theorem in their domain.

The earliest version of NQTHM [9] had no facility for user guidance. The power of that early version of the prover may be very crudely characterized by saying that, starting from Peano’s axioms, and analogous axioms for lists, the prover could not prove anything much more difficult than the associativity of multiplication or the correctness of an insertion sort algorithm.

In order to permit NQTHM to prove theorems harder than these (without ‘‘cheating’’ by adding additional formulas as axioms), the most important step we took was to permit the user to suggest ‘‘lemmas,’’ i.e. intermediate theorems, which would first be proved by NQTHM and then made available for use in subsequent proofs, mainly by the simplifier [10]. Permitting the use of lemmas on the one hand makes NQTHM feel more like a proof checker than a theorem prover, but on the other hand it permits the checking of a very substantial part of elementary number theory [66], even including Gauss's law of quadratic reciprocity (the crown jewel of number theory), and the correctness of some interesting algorithms [16].

5 Our Motivation: Computer System Correctness

Given that NQTHM is not a very ‘‘smart’’ theorem prover, one might well ask why we have kept working on it for so many years! Our main motivation has been to develop NQTHM into a system that can be used in a practical way to check the correctness of computer systems, thereby reducing the frequency of bugs in computer programs.

The idea of proving the correctness of algorithms is at least as old as Euclid's demonstration of the correctness of an algorithm for finding the greatest common divisor of two integers [29]. The idea of correctness proofs is also clearly stated in the classic papers of Goldstine and von Neumann [73] that describe the first von Neumann machine and how to use it. In those papers fifteen programs, including a sort routine, are specified, coded in machine language, and proved correct. Although correctness proofs were undoubtedly constructed by many early programmers, publications about this idea seem rare until the 60s, when McCarthy [56], Floyd [30], Hoare [35], and Burstall [26] described means for proving the correctness of programs written in higher level languages. Subsequently a rather sizable literature on the subject has developed.

Proofs of the correctness of computing systems seem to be much longer, much more tedious, and much more error prone than proofs in ordinary mathematics. The additional length is due to the fact that the computing systems may easily require hundreds of pages of specification, whereas most propositions in mathematics can easily be stated in a few pages -- even if one includes the axiomatization of set theory, analysis, and algebra. The tediousness and error level are perhaps due to this length and also to the fact that many parts of computing systems are mathematically boring.

The idea of mechanically checking proofs of correctness of computing systems has been pursued by many researchers, e.g. those mentioned in the review article [7]. Research on this topic has grown to the extent that there are several research laboratories (e.g. the Computer Science Laboratory of SRI International) and several small companies (e.g. our own Computational Logic, Inc. and Richard Platek's Odyssey Associates, Inc.) which devote a major portion of their efforts to research on this topic.

6 Applications

We and others have used NQTHM to check the correctness of many small programs. However, after many years of effort, we are beginning to see mechanical correctness proofs of entire small computing systems. By far, the most significant application of NQTHM has been to a prove the correctness of a computing system known as the *CLI Stack*, which includes (a) a microprocessor design (FM8502) based on gates and registers [38], (b) an assembler (Piton) [61] that targets FM8502, and (c) a higher level language (micro Gypsy) [76] that targets Piton. We have also seen a proof of correctness of a small operating system kernel (KIT) [2]. Except for the Piton work, all of these projects represent Ph.D. dissertations in computer science which we supervised at the University of Texas. FM8502, Piton, micro Gypsy, and Kit are documented in one place, a special issue of the *Journal of Automated Reasoning* [62].

Another major application of NQTHM is the Ph.D. work of N. Shankar in proof checking Godel's incompleteness theorem [69]. The text of this proof effort is included in the standard distribution of NQTHM, along with Shankar's checking of the Church-Rosser theorem.

On pp. 4-9 of ACLH, we enumerate many other applications of NQTHM, including those in list processing, elementary number theory, metamathematics, set theory, and concurrent algorithms. Descriptions of some of these applications may be found in [16, 66, 12, 21, 17, 67], [68, 69, 20, 60, 28, 51, 37, 52], [13, 14, 15, 22, 77], and also in [1, 31, 32, 33, 40, 75, 3], [48, 44, 41, 42], [39, 45, 23, 24, 25].

Recently colleagues of ours at Computational Logic, Inc., Bill Young and Bill Bevier, have used NQTHM to construct mechanically checked proofs of properties relating to fault-tolerance. A key problem facing the designers of systems which attempt to ensure fault tolerance by redundant processing is how to guarantee that the processors reach agreement, even when one or more processing units are faulty. This problem, called the Byzantine Generals problem or the problem of achieving *interactive consistency*, was posed and solved by Pease, Shostak, and Lamport [64, 50]. They proved that the problem is solvable if and only if the total number of processors exceeds three times the number of faulty processors and devised an extremely clever algorithm (the ‘‘Oral Messages’’ Algorithm) which implements a solution to this problem. Bill Young and Bill Bevier, have just finished developing a machine checked proof of the correctness of this algorithm using NQTHM.

Matt Kaufmann, of Computational Logic, Inc., has made extensive additions to NQTHM, building a system called ‘‘PC-NQTHM’’ on top of NQTHM, which many find more convenient than NQTHM for checking proofs. Information about PC-NQTHM and some extensions and applications may be found in [46, 49, 45, 47, 63, 43, 76]. Among the theorems which Kaufmann has checked with PC-NQTHM are:

- Ramsey's theorem for exponent 2 (both finite and infinite versions), with explicit bound in the finite case [41, 46].
- Correctness of an algorithm of Gries for finding the largest ‘‘true square’’ submatrix of a boolean matrix [40].
- The Cantor-Schroeder-Bernstein theorem [46].
- The correctness of a Towers of Hanoi program.
- The irrationality of the square root of 2.
- Correctness of a finite version of the collapsing function of Cohen forcing.

7 Work in Progress: ACL2

We are currently constructing an entirely new version of our prover. The name of the new system is *A Computational Logic for Applicative Common Lisp*, which might be abbreviated as ‘‘ACL ACL’’ but which we abbreviate as ‘‘ACL2’’. Whereas NQTHM has been available for some time, extensively documented, and widely used, ACL2 is still very much under development. Hence the following remarks are somewhat speculative.

Instead of supporting ‘‘Boyer-Moore logic’’, which reflects an odd mixture of functions vaguely, but not consistently, related to Lisp 1.5 and Interlisp, ACL2 directly supports perfectly and accurately (we hope) a large subset of applicative Common Lisp. That is, ACL2 is to applicative Common Lisp what NQTHM is to the ‘‘Boyer-Moore logic’’, a programming/theorem proving environment for an executable logic of recursive functions.

More precisely, we have identified an applicative subset of Common Lisp and axiomatized it, following Steele’s [72] carefully. Because arrays, property lists, input/output and certain other commonly used programming features are not provided applicatively in Common Lisp (i.e., they all involve the notion of explicit state changes), we axiomatized applicative versions of these features. For example, when one ‘‘changes’’ an array object, one gets a new array object. However, we gave these applicative functions very efficient implementations which are in complete agreement with their axiomatic descriptions but which happen to execute at near von Neumann speeds when used in the normal von Neumann style (in which ‘‘old’’ versions of a modified structure are not accessed). The result is ‘‘applicative Common Lisp’’ which is also an executable mathematical logic.

Like NQTHM, the logic of applicative Common Lisp provides a definitional principle that permits the sound extension of the system via the introduction of recursive functions. Unlike NQTHM, however, functions in applicative Common Lisp may be defined only on a subset of the universe. Like NQTHM, the new logic provides the standard first order rules of inference and induction. However, the axioms are different since, for example, NQTHM and ACL2 differ on what **`(CAR NIL)`** is. Most importantly for the current purposes, we claim that all correct Common Lisps implement applicative Common Lisp directly and that, unlike NQTHM’s logic, applicative Common Lisp is a practical programming language.

ACL2 is a theorem prover and programming/proof environment for applicative Common Lisp. ACL2 includes all of the functionality of NQTHM (as understood in the new setting) plus many new features (e.g., congruence-based rewriting). The source code for ACL2 consists of about 1.5 million characters, all but 43,000 of which are in applicative Common Lisp. That is, 97% of ACL2 is written applicatively in the same logic for which ACL2 proves theorems. The 3% of non-applicative code is entirely at the top-level of the read-eval-print user interface and deals with reading user input, error recovery and interrupts. We expect to implement **`read`** applicatively and limit the non-applicative part of ACL2 to the essential interaction with the underlying Common Lisp host system.

Thus, in ACL2 as it currently stands, the definitional principle is implemented as a function in logic, including the syntax checkers, error handlers, and data base handlers. The entire ‘‘Boyer-Moore theorem prover’’ -- as that term is now understood to mean ‘‘the theorem prover Boyer and Moore have written for ACL2’’ -- is a function in the logic, including the simplifiers, the decision procedures, the induction heuristics, and all of the proof description generators.

The fact that almost all of ACL2 is written applicatively in the same logic for which it is a theorem prover allows the ACL2 source code to be among the axioms in that definitional extension of the logic. The user of the ACL2 system can define functions, combine his functions with those of ACL2, execute them, or prove things about them, in a unified setting. One need only understand one language, Common Lisp, to use the ‘‘logic’’, interact with the system, interface to the system, or modify the system. DEFMACRO can be used to extend the syntax of the language, users can introduce their own front-ends by programming within the logic, and all of the proof routines are accessible to users and have exceptionally clear (indeed, applicative) interfaces. Many new avenues in metatheoretic extensibility are waiting to be explored. We believe we have taken a major step towards the goal of perhaps someday checking the soundness of most of the theorem prover by defining the theorem prover in a formalized logic.

At the time of this writing, we have completely recoded all of the functionality of NQTHM, but have only begun experimentation with proving theorems. However, our preliminary evidence is that there will be no substantial degradation in performance, even though ACL2 is coded applicatively.

8 Conclusions

8.1 Proof Checking as a Mere Engineering Challenge

In our view it seems humanly feasible to write mechanical proof checkers for any part of mathematics and to check mechanically any result in mathematics. There has been much doubt cast on the feasibility of formal proofs, even by such respected authorities as Bourbaki [6]

> But formalized mathematics cannot in practice be written down in full, and therefore we must have confidence in what might be called the common sense of the mathematician ... We shall therefore very quickly abandon formalized mathematics ...

We believe that we have enough practical evidence to extrapolate that mechanical proof checking any mathematical result is feasible, not some mere theoretical possibility which would require a computer the size of the universe. We can make no definite claim about the cost of doing such proof checking, given a suitable proof checker, but we suspect that in the worst case it is somewhere between approximately ten and one hundred times as expensive as doing careful hand proofs at the level of an upper level undergraduate mathematics textbook. In a few areas of mathematics, such as those described by [27] the cost is much less than doing careful hand proofs. We are optimistic that research by top mathematicians will expand the areas in which mechanical theorem-provers are better than most mathematicians.

8.2 Checking the Correctness of Computing Systems

Almost as a corollary to the preceding view, we assert that it is humanly feasible to check mechanically the correctness of computer systems against formal specifications for those systems. Moreover, we believe that the reliability of computing systems could and should be increased significantly by requiring that critical systems be formally specified and that their correctness with respect to those specifications be mechanically checked. Again, we make no definite claim about the cost of doing such certification, but given that, for example, there exist microprocessors that are in control of nuclear weapons, we believe that the cost of doing such checking may well be less than the cost of not doing such checking.

8.3 Formalizing the Real World

Although the correctness of algorithms and even systems is something that is reasonably clearly understood from a mathematical point of view, it remains a major and largely unexplored area of research to formalize the interactions of computing systems with the ‘‘real world.’’ Even correctly formalizing the behavior of a typical industrial microcontroller, with its myriad timers, interrupts, buses, and A/D converters seems to be on the edge of the state of the art of formalization. Any claim that a computing system has been formally proved to interact safely with the world is no better than the accuracy with which the behavior of the world has been formalized. The difficulty of accurately formalizing the behavior of the world does not diminish the fact that typically a very large part of what any computing system is supposed to do (especially the internal workings) can be formally specified, and that part is suitable for scrutiny with formal, mechanical proof attempts.

9 Acknowledgements

We want to express our thanks to a number of people who have contributed significantly towards making NQTHM a successful prover.

The first version of our prover was developed in the amazingly fertile environment of Edinburgh University in the period 1971 to 1974. While working in Bernard Meltzer’s Metamathematics Unit (which then became the Department of Computational Logic), we had the joy of working with such figures as J.A. Robinson, Bob Kowalski, Pat Hayes, Alan Bundy, Aaron Sloman, and Woody Bledsoe. In nearby groups, such as the Department of Machine Intelligence, we found inspiration from the likes of Rod Burstall, Donald Michie, Robin Popplestone, Gordon Plotkin, Michael Gordon, Bruce Anderson, David Warren, Raymond Aubin, Harry Barrow, John Darlington, and Julian Davies. The time and place seemed to be embued with quiet inspiration. It is hard for us to imagine that our prover could have put down its roots anyplace else.

John McCarthy’s influence on our work has been major. His invention of Lisp gave us a language [55, 59] in which to write NQTHM. His papers on proof checking, e.g. [57], and the mathematical theory of computation [58] gave us incentive to write a prover for program verification, reasoning techniques to encode, and sample theorems on which to work. We have mentioned above Woody Bledsoe’s influence on our work in showing how to write heuristic theorem provers similar to ours.

We thank Rod Burstall for his inspiring and elegant paper on structural induction [26]. We thank Burstall, Michie, and Popplestone for use of the POP2 system running on an ICL 4130 on which we coded the earliest version of our prover. POP2 is a Lisp-like language with an Algol-like syntax and many features ahead of its time, including streams and abstract data types, which influenced the design of the shell construct in the NQTHM logic.

At SRI International and Xerox PARC (JSM), we translated our prover into Lisp and made major extensions to it. We owe a debt of thanks to many people there for their support and encouragement, including Robert W. Taylor, Warren Teitelmann, Peter Deutsch, Butler Lampson, Jack Goldberg, Peter Neumann, Karl Levitt, Bernie Elspas, Rob Shostak, Jay Spitzen, Les Lamport, Joe Goguen, Richard Waldinger, Nils Nilsson, and Peter Hart.

We owe our user community a major debt. In particular, we acknowledge the contributions of Bill Bevier, Bishop Brock, S.C. Chou, Ernie Cohen, Jimi Crawford, David Goldschlag, C.H. Huang, Warren Hunt, Myung Kim, David Russinoff, Natarajan Shankar, Mark Woodcock, Matt Wilding, Bill Young, and Yuan Yu. In addition, we have profited enormously from our association with Matt Kaufmann, Hans Kamp, Chris Lengauer, Norman Martin, John Nagle, Carl Pixley, and Bill Schelter. Topher Cooper has the distinction of being the only person to have found an unsoundness in a released version of our system.

We also most gratefully acknowledge the support of our colleagues at the Institute for Computing Science at the University of Texas, now almost all at Computational Logic, especially Don Good and Sandy Olmstead who created and maintained at the Institute a creative and relaxed research atmosphere with excellent computing facilities. In 1986 we moved our entire verification research group (and its atmosphere) off campus and established Computational Logic, Inc.

Notwithstanding the contributions of all our friends and supporters, we would like to make clear that NQTHM is a very large and complicated system that was written entirely by the two of us. Not a single line of Lisp in our system was written by a third party. Consequently, every bug in it is ours alone. Soundness is the most important property of a theorem prover, and we urge any user who finds such a bug to report it to us at once.

The development of our logic and theorem prover has been an ongoing effort for the last 18 years. During that period we have received financial support from many sources. Our work has been supported for over a decade by the National Science Foundation and the Office of Naval Research. Of the many different grants and contracts involved we list only the latest: NSF Grant DCR-8202943, NSF Grant DCR81-22039, and ONR Contract N00014-81-K-0634. We are especially grateful to NSF, ONR, and our technical monitors there, Tom Keenan, Bob Grafton, and Ralph Wachter, for years of steady support and encouragement.

The development of our prover is currently supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 6082 and by the Office of Naval Research, ONR Contract N00014-88-C-0454.

We have received additional support over the years from the following sources, listed chronologically: Science Research Council (now the Science and Engineering Research Council) of the United Kingdom, Xerox, SRI International, NASA, Air Force Office of Scientific Research, Digital Equipment Corporation, the University of Texas at Austin, the Venture Research Unit of British Petroleum, Ltd., and IBM.

We thank Bill Schelter for the numerous suggestions he has made for improving the performance of NQTHM under Austin-Kyoto Common Lisp.

Thanks to Anne Boyer for editing this and other writings.

Finally, we wish to express one negative acknowledgement. The research group assembled at Edinburgh in the early 70’s was scattered to the winds by the ‘‘Lighthill Report,’’ the devastatingly negative review of artificial intelligence in Britain conducted

by Sir James Lighthill. If computing becomes the dominant branch of both science and engineering, as seems possible, we hope that renowned computer scientists, if asked, will take the greatest care to review new developments in physics with humility, not arrogance, and not attempt to quash new developments that do not fit into old paradigms of science.

References

1. W. Bevier. *A Verified Operating System Kernel*. Ph.D. Th., University of Texas at Austin, 1987.

2. W. R. Bevier. "Kit and the Short Stack". *Journal of Automated Reasoning 5*, 4 (1989), 519-530.

3. William Bevier, Matt Kaufmann, and William Young. Translation of a Gypsy Compiler Example into the Boyer-Moore Logic. Internal Note 169, Computational Logic, Inc., January, 1990.

4. W.W. Bledsoe. "Splitting and Reduction Heuristics in Automatic Theorem Proving". *Artificial Intelligence 2* (1971), 55-77.

5. W. Bledsoe, R. Boyer, and W. Henneman. "Computer Proofs of Limit Theorems". *Artificial Intelligence 3* (1972), 27-60.

6. N. Bourbaki. *Elements of Mathematics.* Addison Wesley, Reading, Massachusetts, 1968.

7. R. S. Boyer and J S. Moore. "Program Verification". *Journal of Automated Reasoning 1*, 1 (1985), 17-23.

8. R. S. Boyer, D. M. Goldschlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First Order Logic, Report 44. Computational Logic, 1717 W. 6th St., Austin, Texas, 78703, U.S.A., 1989. To appear in the proceedings of the 1989 Workshop on Programming Logic, Programming Methodology Group, University of Goteborg.

9. R. S. Boyer and J S. Moore. "Proving Theorems about LISP Functions". *JACM 22*, 1 (1975), 129-144.

10. R. S. Boyer and J S. Moore. A Lemma Driven Automatic Theorem Prover for Recursive Function Theory. Proceedings of the 5th Joint Conference on Artificial Intelligence, 1977, pp. 511-519.

11. R. S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

12. R. S. Boyer and J S. Moore. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

13. R. S. Boyer and J S. Moore. A Verification Condition Generator for FORTRAN. In *The Correctness Problem in Computer Science*, R. S. Boyer and J S. Moore, Eds., Academic Press, London, 1981.

14. R. S. Boyer and J S. Moore. The Mechanical Verification of a FORTRAN Square Root Program. SRI International, 1981.

15. 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.

16. R. S. Boyer and J S. Moore. "Proof Checking the RSA Public Key Encryption Algorithm". *American Mathematical Monthly 91*, 3 (1984), 181-189.

17. R. S. Boyer and J S. Moore. "A Mechanical Proof of the Unsolvability of the Halting Problem". *JACM 31*, 3 (1984), 441-458.

18. R. S. Boyer and J S. Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.

19. R. S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study with Linear Arithmetic. In *Machine Intelligence 11*, Oxford University Press, 1988.

20. R. S. Boyer and J S. Moore. "The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover". *Journal of Automated Reasoning 4* (1988), 117-172.

21. R. S. Boyer and J S. Moore. A Mechanical Proof of the Turing Completeness of Pure Lisp. In *Automated Theorem Proving: After 25 Years*, W.W. Bledsoe and D.W. Loveland, Eds., American Mathematical Society, Providence, R.I., 1984, pp. 133-167.

22. R. S. Boyer, M. W. Green and J S. Moore. The Use of a Formal Simulator to Verify a Simple Real Time Control Program. In D. Gries, et. al, Ed., *Beauty Is Our Business*, Springer, 1990. To Appear.

23. A. Bronstein and C. Talcott. String-Functional Semantics for Formal Verification of Synchronous Circuits, Report No. STAN-CS-88-1210. Computer Science Department, Stanford University, 1988.

24. A. Bronstein. *MLP: String-functional semantics and Boyer-Moore mechanization for the formal verification of synchronous circuits*. Ph.D. Th., Stanford University, 1989.

25. A. Bronstein and C. Talcott. Formal Verification of Synchronous Circuits based on String-Functional Semantics: The 7 Paillet Circuits in Boyer-Moore. C-Cube 1989 Workshop on Automatic Verification Methods for Finite State Systems. LNCS 407, 1989, pp. 317-333.

26. R. Burstall. "Proving Properties of Programs by Structural Induction". *The Computer Journal 12*, 1 (1969), 41-48.

27. S. Chou. *Mechancial Geometry Theorem Proving.* Reidel, 1988.

28. Benedetto Lorenzo Di Vito. *Verification of Communications Protocols and Abstract Process Models*. Ph.D. Th., University of Texas at Austin, 1982.

29. T. L. Heath (translation and commentary). *The Thirteen Books of Euclid's Elements.* Dover, New York , 1908. p. 298, Vol 2., i.e. Proposition 2, Book VII.

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

31. David M. Goldschlag. "Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover". *IEEE Transactions on Software Engineering* (September 1990). To appear.

32. David M. Goldschlag. Mechanizing Unity. In *Proceedings of the IFIP TC2/WG2.3 Working Conference on Programming Concepts and Methods,* M. Broy and C. B. Jones, Eds., Elsevier, Amsterdam, 1990.

33. David M. Goldschlag. "Proving Proof Rules: A Proof System for Concurrent Programs". *Compass '90* (June 1990).

34. R. L. Goodstein. *Recursive Number Theory.* North-Holland Publishing Company, Amsterdam, 1964.

35. C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". *Comm. ACM 12,* 10 (1969), 576-583.

36. L. Hodes. Solving Problems by Formula Manipulation. Proc. Second Inter. Joint Conf. on Art. Intell., The British Computer Society, 1971, pp. 553-559.

37. C.-H. Huang and C. Lengauer. "The Automated Proof of a Trace Transformation for a Bitonic Sort". *Theoretical Computer Science 1*, 46 (1986), 261-284.

38. W. A. Hunt. "Microprocessor Design Verification". *Journal of Automated Reasoning 5,* 4 (1989), 429-460.

39. Matt Kaufmann. A Formal Semantics and Proof of Soundness for the Logic of the NQTHM Version of the Boyer-Moore Theorem Prover. Internal Note 229, Institute for Computing Science, University of Texas at Austin, February, 1987.

40. Matt Kaufmann. A Mechanically-checked Semi-interactive Proof of Correctness of Gries's Algorithm for Finding the Largest Size of a Square True Submatrix. Internal Note 236, Institute for Computing Science, University of Texas at Austin, October, 1986.

41. Matt Kaufmann. An Example in NQTHM: Ramsey's Theorem. Internal Note 100, Computational Logic, Inc., November, 1988.

42. Matt Kaufmann. Boyer-Moore-ish Micro Gypsy and a Prototype Hardware Expander. Internal Note 73, Computational Logic, Inc., August, 1988.

43. Matt Kaufmann. A Mutual Recursion and Dependency Analysis Tool for NQTHM. Internal Note 99, Computational Logic, Inc., 1988.

44. Matt Kaufmann. A User's Manual for RCL. Internal Note 157, Computational Logic, Inc., October, 1989.

45. Matt Kaufmann and Matt Wilding. A Parallel Version of the Boyer-Moore Prover. Tech. Rept. 39, Computational Logic, Inc., February, 1989.

46. Matt Kaufmann. DEFN-SK: An Extension of the Boyer-Moore Theorem Prover to Handle First-Order Quantifiers. Tech. Rept. 43, Computational Logic, Inc., 1717 W. 6th St, Suite 290, Austin, Texas, June, 1989.

47. Matt Kaufmann. Addition of Free Variables to an Interactive Enhancement of the Boyer-Moore Theorem Prover. Tech. Rept. 42, Computational Logic, Inc., Austin, Texas, May, 1989.

48. Matt Kaufmann. A Mechanically-checked Correctness Proof for Generalization in the Presence of Free Variables. Tech. Rept. 53, Computational Logic, Inc., Austin, Texas, March, 1990.

49. Matt Kaufmann. An Integer Library for NQTHM. Internal Note 182, Computational Logic, Inc., March, 1990.

50. Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". *ACM TOPLAS 4,* 3 (July 1982), 382-401.

51. C. Lengauer. "On the Role of Automated Theorem Proving in the Compile-Time Derivation of Concurrency". *Journal of Automated Reasoning 1*, 1 (1985), 75-101.

52. C. Lengauer and C.-H. Huang. A Mechanically Certified Theorem about Optimal Concurrency of Sorting Networks. Proc. 13th Ann. ACM Symp. on Principles of Programming Languages, 1986, pp. 307-317.

53. D. Loveland. *Automated Theorem Proving: A Logical Basis.* North Holland, Amsterdam, 1978.

54. J. McCarthy. "Recursive Functions of Symbolics Expressions and their Computation by Machine". *Communications of the Association for Computing Machinery 3*, 4 (1960), 184-195.

55. J. McCarthy. *The Lisp Programmer's Manual.* M.I.T. Computation Center, 1960.

56. J. McCarthy. Towards a Mathematical Science of Computation. Proceedings of IFIP Congress, 1962, pp. 21-28.

57. J. McCarthy. Computer Programs for Checking Mathematical Proofs. Recursive Function Theory, Proceedings of a Symposium in Pure Mathematics, Providence, Rhode Island, 1962, pp. 219-227.

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

59. J. McCarthy, et al. *LISP 1.5 Programmer's Manual.* The MIT Press, Cambridge, Massachusetts, 1965.

60. J S. Moore. "A Mechanical Proof of the Termination of Takeuchi's Function". *Information Processing Letters 9*, 4 (1979), 176-181.

61. J. S. Moore. "A Mechanically Verified Language Implementation". *Journal of Automated Reasoning 5*, 4 (1989), 461-492.

62. J. S. Moore, et. al. "Special Issue on System Verification". *Journal of Automated Reasoning 5*, 4 (1989), 409-530.

63. Matt Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover. Tech. Rept. 19, Computational Logic, Inc., Austin, Texas, May, 1988.

64. Marshall Pease, Robert Shostak, and Leslie Lamport. "Reaching Agreement in the Presence of Faults". *JACM 27*, 2 (April 1980), 228-234.

65. J. A. Robinson. "A Machine-oriented Logic Based on the Resolution Principle". *JACM 12*, 1 (1965), 23-41.

66. David M. Russinoff. "An Experiment with the Boyer-Moore Theorem Prover: A Proof of Wilson's Theorem". *Journal of Automated Reasoning 1*, 2 (1985), 121-139.

67. N. Shankar. "Towards Mechanical Metamathematics". *Journal of Automated Reasoning 1*, 4 (1985), 407-434.

68. N. Shankar. A Mechanical Proof of the Church-Rosser Theorem. Tech. Rept. ICSCA-CMP-45, Institute for Computing Science, University of Texas at Austin, 1985.

69. N. Shankar. *Proof Checking Metamathematics*. Ph.D. Th., University of Texas at Austin, 1986.

70. J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, Ma., 1967.

71. T. Skolem. The Foundations of Elementary Arithmetic Established by Means of the Recursive Mode of Thought, without the Use of Apparent Variables Ranging over Infinite Domains. In *From Frege to Godel,* J. van Heijenoort, Ed., Harvard University Press, Cambridge, Massachusetts, 1967.

72. G. L. Steele, Jr. *Common Lisp The Language.* Digital Press, 30 North Avenue, Burlington, MA 01803, 1984.

73. J. von Neumann. *John von Neumann, Collected Works, Volume V.* Pergamon Press, Oxford, 1961.

74. L. Wos, et al. "The concept of demodulation in theorem proving". *Journal of the ACM 14* (1967), 698-709.

75. Matt Kaufmann and William D. Young. Comparing Gypsy and the Boyer-Moore Logic for Specifying Secure Systems. Institute for Computing Science, University of Texas at Austin, May, 1987. ICSCA-CMP-59.

76. W. D. Young. "A Mechanically Verified Code Generator". *Journal of Automated Reasoning 5*, 4 (1989), 493-518.

77. Yuan Yu. "Computer Proofs in Group Theory". *Journal of Automated Reasoning* (1990). To appear.

Table of Contents