Noname manuscript No.
(will be inserted by the editor)

Milestones from The Pure Lisp Theorem Prover to ACL2

Draft 4 – 5 March, 2019

J Strother Moore

the date of receipt and acceptance should be inserted later

Abstract We discuss the evolutionary path from the Edinburgh Pure Lisp Theorem Prover of the early 1970s to today's ACL2. Among the milestones are the adoption of a first-order subset of a programming language as a logic; the analysis of recursive definitions to guess appropriate mathematical induction schemes; the use of simplification in inductive proofs; the incorporation of rewrite rules derived from user-suggested lemmas; the generalization of that idea to allow the user to affect other proof techniques soundly; the recognition that evaluation efficiency is paramount so that formal models can serve as prototypes and the logic can be used to reprogram the system; use of the system to prove extensions correct; the incorporation of decision procedures; the provision of hierarchically structured libraries of previously certified results to configure the prover; the provision of system programming features to allow verification tools to be built and verified within the system; the release of many verified collections of lemmas supporting floating point, programming languages, and hardware platforms; a verified "bit-bashing" tool exploiting verified BDD and checked external SAT procedures; and the provision of certain higher-order features within the first-order setting. As will become apparent, some of these milestones were suggested or even prototyped by users. Some additional non-technical aspects of the project are also critical. Among these are a devotion to soundness, good documentation, freely available source code, production of a system usable by industry, responsiveness to user needs, and a dedicated, passionate, and brilliant user community.

Keywords theorem proving, hardware, software, verification, functional programming, Lisp, induction, rewriting, reflection, decision procedures

Department of Computer Science
University of Texas at Austin
E-mail: moore@cs.utexas.edu

1 About ACL2

This paper is about the evolution of the Edinburgh Pure Lisp Theorem Prover (PLTP) into today's ACL2. PLTP is an extremely simple prover for recursive functions and inductive data types, supporting Pure Lisp as its logic.[1] It was developed by Robert S. Boyer and me between 1971 and 1973. It was the first of its kind. ACL2, which stands for *A* *C*omputational *L*ogic for *A*pplicative *C*ommon *L*isp, is an "industrial strength" prover in routine use in industry. It was initially designed by Boyer and me in 1989, but for the past 26 years has been the still-evolving product of Matt Kaufmann and me. To justify talking about the journey to ACL2, let me first explain our destination.

ACL2 is a theorem prover for reasoning about recursive functions and inductively constructed objects. Its logic is an extension of an applicative (functional) subset of ANSI standard Common Lisp [55]. A user of ACL2 can prototype a digital artifact like a circuit design or an algorithm in an ANSI standard programming language, efficiently run the formal model on concrete data, and verify properties by reasoning mechanically and directly about the prototype. Mathematical induction is common because most models involve recursive functions over inductively constructed domains. It is distributed without cost under a very lenient 3-clause BSD license and in source code form, with extensive documentation and a massive library of reusable certified results.

Among industrial artifacts whose properties have been verified with ACL2 are a Motorola digital signal processor [18], the floating point arithmetic in the Advanced Micro Devices (AMD) Athlon [51], the Rockwell Collins AAMP7G cryptographic chip (providing the basis of an NSA MILS certification) [24], critical security related properties of the J2ME JVM [38], many components of the Centaur Nano media unit [28], and some x86 machine code programs [23]. The last illustrates an important feature of ACL2: despite being a logic with good mechanical proof support it is a programming language that executes efficiently. The x86 model describes both the system- and user-levels of the architecture and has been used to prove some machine code programs correct. But it is also a useful simulator, running user-level machine code at 3.3 million instructions per second and system level code (where page tables complicate execution) at 912,000 instructions per second.

Increasingly the most impressive accomplishments of ACL2 are being carried out by industrial users, often working on proprietary designs. These users are often able only to share the basic mathematical infrastructure they have built, by contributing certified *books* of definitions and lemmas to the repository. For example, David Russinoff has done floating point proofs in ACL2 for AMD, Intel and ARM using a comprehensive floating point library, named "rtl/rtl11" which he has shared with the community. Furthermore, he has written a conventional but highly detailed mathematics monograph on the results he makes available [52].[2]

[1] I write in the present tense, even about provers that are no longer in use, because I regard them as mathematical artifacts and algorithms, not just running code.

[2] See Section 8 for details on how to access shared results in ACL2.

But most ACL2 proofs today are carried out in industry and do not result in publications. For example, Centaur Technology, Inc., a subsidiary of VIA, designs high performance, low-cost x86 compatible microprocessors. Centaur runs thousands of ACL2 proof scripts nightly to check tens of thousands of properties of its evolving design ("bugs introduced today are detected tonight and fixed tomorrow"). By the standards of formal logic, these designs are huge. The VIA Nano floating-point adder is actually composed of four adders so that the unit can simultaneously add four pairs of single-precision numbers, two pairs of double-precision numbers, or one pair of extended-precision numbers. The unit can complete four floating-point additions in two steps (clock cycles), and it is pipelined so it can produce four results every clock cycle. The unit's description involves 33,700 lines of Verilog in 680 modules; its implementation involves 432,322 transistors. The floating-point adder is part of a larger media unit that has 1074 inputs, including 26 clock inputs, and 374 output signals. The unit performs over 100 other operations – and many (> 1000) inputs have to be set properly to force it to perform the additions to be verified.

Much of the ACL2 infrastructure Centaur has produced [54] is available in the Community Books including, for example, a Verilog to ACL2 translator, but individual theorems about Centaur designs are not generally published. By sharing the infrastructure, Centaur has allowed other people and organizations, such as Oracle, to build on it.

So how did we get here? It is not unreasonable to think of ACL2 as just the latest version of PLTP! But in any long journey there are milestones and in this paper I lay out the ones I consider most important.

2 The Relevant Provers

There are four main provers in this evolutionary sequence: PLTP, "Thm," which is what I shall call the prover described in the 1979 book *A Computational Logic* [8], Nqthm, and ACL2. But it is wrong to think of any of these provers as having been stable. Each was under constant development and it is difficult to say when, for example, PLTP became Thm or Thm became Nqthm. But roughly speaking, PLTP was in active use for about 5 years, from 1972 to 1976. Thm was used about that long too. Nqthm was the first to have a user community outside of theorem proving researchers and it was in active use from the early-1980s until the mid-1990s. Because of the user community, we maintained and even used Nqthm ourselves for about 5 years after ACL2 had become our main focus. ACL2 was started in August, 1989, and is still in active use these 30 years later, far longer than its predecessors. But it too has been under constant development: forty two versions have been released since 1996, about one every 6 months. These different versions are not mere updates to fix bugs and cope with platform changes but often include major new logical, proof, or programming features. Broadly speaking these versions of ACL2 have been "backwards compatible" although sometimes script changes were required to get "old" theorems to be proved anew. The major impetus for the constant development of ACL2 and, counter-intuitively, its relative stability compared to its predecessors, has been the dependence of industrial users on the tool and the constantly escalating challenges of the problems they try to tackle with ACL2.

No one but the authors (Boyer and Moore for PLTP, Thm, and Nqthm, and Boyer, Kaufmann and Moore for ACL2) modifies the code for these provers, although the systems are distributed in source code form. Users can and do make suggestions, find bugs, and even contribute prototypes of improvements.

None of our provers are foundational: they do not produce formal proofs. Instead they do computations that we believe guarantee that a formal proof exists. The easiest to understand example is the contrast between a foundational prover that proves tautologies by constructing a proof in terms of *modus ponens* and other propositional rules versus one that uses, say, a decision procedure like BDDs or SAT.

Our provers have all had soundness bugs from time to time, mainly because of programming errors but occasionally because of flawed logical thinking on our parts. But such bugs are relatively rare, generally quite difficult to exploit, and are taken very seriously by users and the implementors. We announce them and fix them as soon as possible.

All four provers use some version of applicative (functional) Lisp as the logic. All are untyped in the sense that Lisp is untyped: the syntax imposes no type constraints on functions and their results, but each Lisp provides several types of objects. The syntax is free of quantifiers. All the Lisps are formalized in first-order logic with induction. Functions in the logics can be run on concrete input, an especially useful feature when the functions model a complicated computational artifact because models have dual use as both prototypes and formal objects about which properties can be proved.

The expressive weakness of the logics actually plays a role in the success of the provers. To prove $\forall n \exists x : \psi(n, x)$ the user must define a function, f, to construct a suitable x from n and then prove $\psi(n, f(n))$. The definition of f gives the provers a big hint. As Grant Passmore has said [private communication], "some provers extract programs from proofs, but ACL2 extracts proofs from programs."

All are an *ad hoc* mix of proof procedures governed by heuristics. Completeness is of little concern; the theories are undecidable. We have always prized efficient automation, especially for theorems humans find "obvious."

For the purposes of this paper I tend to use ACL2's Common Lisp notation, which I assume readers will understand or figure out from the narrative.

3 Organization

Section 1 gives some information about ACL2 to try to justify the reader's investment in learning about its evolution. Section 2 identifies the four provers discussed here and summarizes their commonalities. Section 4 sets the scene for program verification and theorem proving research in the late 1960s.

Section 5 describes our starting point, the Edinburgh Pure Lisp Theorem Prover, PLTP. In this and each subsequent section I discuss the logic, the prover, some landmark theorems, and the code base. Section 6 describes the evolution from PLTP to Thm. Section 7 describes the evolution to the Nqthm prover in the 1980s. Section 8 describes ACL2. The description includes the same subsections mentioned earlier plus a description of "programming" features. Because ACL2 is a "living" system that is extensively documented, this section has been condensed by the frequent reference to specific online documentation topics the reader may freely browse. Section 9 sums up and in Section 10 I try to describe the scale of my debt to my colleagues.

4 The Milieu

In 1963, John McCarthy published "A Basis for a Mathematical Theory of Computation" [41], in which he used Lisp to specify and prove some program properties, sometimes by modeling the programs' behavior with functions. In 1967, Robert Floyd published "Assigning Meaning to Programs," which described the inductive assertion style of program verification. In 1969, James C. King's PhD dissertation, *A Program Verifier* [36], was published, showing a verification condition generator connected to an *ad hoc* prover for arithmetic. (Arithmetic provers of the time suffered from their inability to handle arithmetic functions not built-in, e.g., mod and gcd, often needed in specifications.) Also in 1969, Tony Hoare published his seminal "An Axiomatic Basis for Computer Programming." Thus, by the end of the 1960s, the formal verification of computer programs was an established research area. But the work was focused mainly on program semantics, i.e., how to transform a formally annotated program into formulas that, if theorems, would establish that the program met its specification. It was left to the theorem proving researchers to find a way to prove these formulas.

Meanwhile, research on heuristically based theorem proving was shaken by Hao Wang's 1960 paper "Toward Mechanical Mathematics" [57] that showed a decision procedure that had proved "all [propositional] theorems, totaling near 400, of *Principia Mathematica*." In 1965, Alan Robinson published the seminal "A Machine-Oriented Logic Based on the Resolution Principle," [50] in which he introduced the notion of the most general unifier and a complete uniform proof procedure for first-order predicate calculus. By the end of the 1960s, almost all theorem proving research – upon which the success of program verification depended – was focused on uniform proof procedures and resolution in particular.

One notable exception was Woody Bledsoe at the University of Texas at Austin, who in the late 1960s explored interactive heuristic provers [2] as well as resolution. His 1971 paper "Splitting and Reduction Heuristics in Automatic Theorem Proving" [1] described a heuristically-based set theory prover that would do simple inductions on natural numbers if the theorem was of the form $\forall n \in \mathbf{N} \ldots$.

Turning to the more personal, in September, 1970, having completed a bachelors degree in mathematics at MIT, I enrolled in the PhD program at the University of Edinburgh in the Department of Machine Intelligence.[3] Initially my adviser was Donald Michie but as I got more interested in theorem proving I switched advisers to Rod Burstall. As an undergraduate I had supported myself as a programmer[4] and I continued to do so in Edinburgh. I got a job programming for Bob Kowalski in the Metamathematics Unit with the task of implementing an SL Resolution prover [37].

Simultaneously, Bob Boyer was finishing his PhD at the University of Texas at Austin, working under Bledsoe. In fact, they were both at MIT, where Bledsoe spent a sabbatical. Boyer's dissertation, which was completed during my first year at Edinburgh, was on Locking Resolution [14]. But Boyer also worked with Bledsoe on his non-resolution prover. In September, 1971, Boyer came to the Metamathematics Unit for post-doctoral research into resolution. We shared an office because we were both Americans (indeed, Texans) and kept the office warmer than others.

The programming environment was, to say the least, challenging by today's standard. We coded in POP-2, a imperative Lisp-like programming language with an Algol-like syntax developed at the University of Edinburgh by Robin Popplestone and Rod Burstall. POP-2 ran on an ICL 4130 processor with 64K bytes of 2 microsecond memory shared between 8 interactive jobs controlled by teletype input from the users. File input and output was by paper tape and line printer.

Memory limitations made the SL prover quite limited, and my attempts to save space by representing clauses in a compressed string format slowed it down. Boyer and I soon discovered a structure shared representation of clauses that made it more practical [5] and which also tended to highlight the similarities between linear resolution proofs and conventional stack-based implementations of subroutine calls – a serendipitous observation reinforcing the emerging idea of "programming in predicate calculus."

Given the milieu, as well as our abiding interest in programming, we tended to focus our theorem proving challenges on proving properties of programs. We started using the SL prover to interpret programs in a predicate calculus programming language of my design called Baroque (after a peculiar variant of chess taught to us by Steve Crocker) and to prove theorems about them.[5]

[3] The Machine Intelligence Department was co-located at Hope Park Square with the Metamathematics Unit, and both became founding members of Edinburgh's School of Artificial Intelligence in 1970.

[4] I started coding in FORTRAN in 1964, learned Lisp from Minsky at MIT, and worked successively in many different languages in the Laser Research Group (solving differential equations), IBM Systems (page fault simulation), TRW (debugging the Apollo onboard software controlling Lunar Orbit Insertion), and State Street Bank and Trust (mutual funds).

[5] Also during our time in Edinburgh we developed a text editor, called the 77-editor [4], named for the disk track it was on. The 77-editor was a command line editor with POP-2 as the command language. Commands were based loosely on MIT's Teco. But the 77-editor, which was used in the School of AI in Edinburgh for about 10 years, had a really important feature: the text to be edited was represented in a data structure that kept most

Here is a Baroque program for computing the length of a list taken from page 71 of my dissertation [42].

```
LEN1: (LENGTH NIL) -> 0;
LEN2: (LENGTH (CONS X Y)) -> Z
      WHERE
      (LENGTH Y) -> U;
      (ADD U 1) -> Z;
      END;
```

We formalized a simple Pure Lisp subset in Baroque so that `LENGTH` could be defined with a three-place if-then-else we called `COND`:

```
LENGTH: (LENGTH X) -> U
        WHERE
        (COND X
              (ADD1 (LENGTH (CDR X)))
              0) -> U;
        END;
```

Our SL prover could prove such theorems as "there is a list whose `LENGTH` is 2." But we could not prove that `(LENGTH (APPEND A B))` is the sum of the `LENGTH`s of `A` and `B`. That proof requires induction. This led us, by early 1972, to begin constructing a prover that could prove such theorems. Following McCarthy's lead, we adopted Pure Lisp as our logic. Following Bledsoe's, we studied the way we proved simple theorems about Lisp and just tried to implement it.

5 PLTP

5.1 Logic

We adopted the simplest version of Lisp that would allow us to experiment with proofs by induction. We formalized it in untyped first-order logic with the usual rules, e.g., *modus ponens*, instantiation, etc., plus induction, because that was both familiar and adequate. Five function symbols are axiomatized in PLTP: `IF`, `EQUAL`, `CONS`, `CAR`, and `CDR`.[6] `(CAR (CONS X Y))` is axiomatized to be `X` and `(CAR NIL)` is `NIL`; `CDR` is treated similarly. There is one constant symbol, `NIL`.

of the document on disk and which had a very small memory footprint. The representation facilitated undoing and allowed for meta-data associated with changes. I explained our data structure to Charles Simonyi when I moved to Xerox Park and it was used by him in his implementation of the Bravo text editor ([26], p. 199). I maintained an Interlisp package implementing it so Charles could experiment and request new features long after I left PARC in 1976 [44]. Subsequently, Charles used the representation in Microsoft Word and it is still in use in Word today [private communication].

[6] In my dissertation, which was written in the summer of 1973, the function I here call "`IF`" was named "`COND`," but it was just a 3-place if-then-else. By October, 1973, we had changed the name "`COND`" to "`IF`," as evidenced by the file named `REWRITE` in Listing A of the PLTP Archive discussed below.

T was just an abbreviation for (CONS NIL NIL). PLTP allows user-defined recursive functions. Propositional connectives AND, OR, NOT and IMPLIES are just functions defined in terms of IF. Natural numbers are represented as lists of NILs, which may be abbreviated with numerals. The arithmetic functions are defined recursively *a la* Peano.

5.2 Prover

We aimed for complete automation. No user guidance was permitted and only the axioms and function definitions were available. This meant that the system had to be able to do several nested inductions to carry out many proofs.

The basic proof strategy of PLTP is as follows. First, simplification is tried. Simplification in PLTP consists of symbolic evaluation (rewriting with axioms, including recursive definitions, with heuristics to stop recursions), normalization of IF-expressions, and reduction (eliminating branches containing contradictory tests). The primary heuristic for stopping recursion is to reject the expansion of a function call if it introduces, in the simplified recursive calls, CAR/CDR expressions of variables.

If simplification does not prove the formula, equality hypotheses (EQUAL tests in IFs) are used heuristically to replace one term by another in the true branch of the IF. Heuristics control this substitution and often discard the equality after "using" it, generalizing the goal.

Finally, explicit generalization of common subterms is tried, replacing certain subterms by new variables. The new variables are constrained by additional type hypotheses added to the conjecture.

Those "type hypotheses" are derived automatically by inspection of the term being replaced and are expressed in terms of new predicates introduced by that inspection. The predicates attempt to capture the "output type" of the function symbol of the term being generalized. The most common such invention is "this term always produces a number" but can, for example, "invent" the concept of "even number" to describe the output of a function that doubles its input.

The last proof technique tried is induction. By this point in the process the conjecture has been maximally simplified, any previous induction hypotheses have been used, and (perhaps) a sufficiently general goal has been produced.

The induction heuristic first runs the symbolic Lisp evaluator to record (in a data structure called "the bomb list") recursive calls on CAR and CDR components of variables in the conjecture. These are recursions that "bomb out" because they introduce talk of new components. Each such call "suggests" an induction designed to "catch" the recursion and specify, via the induction hypothesis, the recursive results. For example if the bomb list contains a recursion on (CDR (CDR X)) then the following induction scheme is suggested, where (ψ X) is the conjecture to be proved:

```
(AND (ψ NIL)
     (ψ (CONS X1 NIL))
(IMPLIES (ψ X) (ψ (CONS X2 (CONS X3 X)))))
```

so that the recursive call in the conclusion of the inductive step can be simplified in terms of X2, X3, and X, without introducing CAR/CDR terms.

If several suggestions share variables and treat shared variables compatibly they are merged. Surviving suggestions are rated by evaluating the conjecture on symbolic data computed from each induction hypothesis to see which "catches" the most recursions. Ties are broken by favoring induction on variables not yet inducted upon. Then the base case(s) and induction step are conjoined into a single formula as above and the whole process starts over with simplification.

The entire subgoal structure of the proof is preserved in a single IF-expression produced by expanding the propositional connectives by symbolic evaluation. All the heuristics were designed to support inductive proofs. For example, evaluation stops recursive expansion because of the need to introduce new CAR/CDR terms, but induction uses evaluation to put sufficient structure into the induction conclusion to allow terms to expand and be "caught" by the induction hypothesis. The heuristic use of EQUAL was designed to use induction hypotheses that are equalities, substituting into the induction conclusion so as to eliminate the side of the equality "caught" by the induction and thus to re-express the goal entirely in terms of the side that was not. Generalization similarly eliminates terms shared between the induction hypothesis and the induction conclusion, i.e., the terms "caught" by the induction. Our aim was that PLTP express each new goal in terms suitable for a different induction.

The symbolic evaluator is, of course, capable of fully evaluating a ground term and so it was possible to run our Lisp subset on simple examples.

5.3 Landmark Theorems

The landmark theorems first proved fully automatically by PLTP include various simple arithmetic identities, the associativity of list concatenation, the fact that the reverse of the reverse of a list is the original list, and the correctness of insertion sort (i.e., that sort produces an ordered result containing the same number of occurrences of every element). The apparently simple fact that Peano multiplication is associative requires three inductions and "discovers" (via heuristic equality substitution and generalization) that multiplication distributes over addition and that addition is associative.

Below is PLTP's output on the conjecture that insertion SORT produces ORDERED output. SORT uses ADDTOLIS to add a new element to a list and uses LTE ("less than or equal") to compare elements.

In order to show exactly what PLTP did in this proof, I display its actual output from July, 1973, below. COND is used where we would subsequently use IF. Square brackets are used to delimit terms where we now use parentheses.[7]

[7] We apparently switched back and forth between square brackets and parentheses. Boyer's pretty printer memo (see the PLTP archive mentioned below), with code dating

Everywhere else in this article I use ACL2 notation for all the provers even though each supported a different version of Lisp.

```
THEOREM TO BE PROVED:
[ORDERED [SORT A]]

MUST TRY INDUCTION.

INDUCT ON A.

THE THEOREM TO BE PROVED IS NOW:
[AND [ORDERED [SORT NIL]]
     [IMPLIES [ORDERED [SORT A]] [ORDERED [SORT [CONS A1 A]]]]]

WHICH IS EQUIVALENT TO:
[COND [ORDERED [SORT A]] [ORDERED [ADDTOLIS A1 [SORT A]]] T]

GENERALIZE COMMON SUBTERMS BY REPLACING [SORT A] BY GENRL1.

THE GENERALIZED TERM IS:
[COND [ORDERED GENRL1] [ORDERED [ADDTOLIS A1 GENRL1]] T]

MUST TRY INDUCTION.

(SPECIAL CASE REQUIRED)

INDUCT ON GENRL1.

THE THEOREM TO BE PROVED IS NOW:
[AND
 [COND [ORDERED NIL] [ORDERED [ADDTOLIS A1 NIL]] T]
 [AND
    [COND [ORDERED [CONS GENRL11 NIL]]
    .     [ORDERED [ADDTOLIS A1 [CONS GENRL11 NIL]]]
    .     T]
    [IMPLIES [COND [ORDERED [CONS GENRL12 GENRL1]]
                   [ORDERED [ADDTOLIS A1 [CONS GENRL12 GENRL1]]]
                   T]
             [COND [ORDERED [CONS GENRL11 [CONS GENRL12 GENRL1]]]
                   [ORDERED [ADDTOLIS A1 [CONS GENRL11 [CONS GENRL12 GENRL1]]]]
                   T]]]]

WHICH IS EQUIVALENT TO:

[COND [LTE A1 GENRL11] T [LTE GENRL11 A1]]

MUST TRY INDUCTION.

INDUCT ON GENRL11 AND A1.

THE THEOREM TO BE PROVED IS NOW:
[AND [AND [COND [LTE A1 NIL] T [LTE NIL A1]]
          [COND [LTE NIL GENRL11] T [LTE GENRL11 NIL]]]
     [IMPLIES [COND [LTE A1 GENRL11] T [LTE GENRL11 A1]]
```

from April, 1973, shows "a typical formula from our theorem prover" printed with parentheses. But the July, 1973, proof output used square brackets and then my September, 1973, dissertation used parentheses.

```
              [COND [LTE [CONS A11 A1] [CONS GENRL111 GENRL11]]
                    T
                    [LTE [CONS GENRL111 GENRL11] [CONS A11 A1]]]]]

WHICH IS EQUIVALENT TO:
T

FUNCTION DEFINITIONS:
[SORT [LAMBDA [X] [COND X [ADDTOLIS [CAR X] [SORT [CDR X]]] NIL]]]

[ORDERED
 [LAMBDA
  [X]
  [COND
      X
      [COND [CDR X] [COND [LTE [CAR X] [CAR [CDR X]]] [ORDERED [CDR X]] NIL] T]
      T]]]

[LTE [LAMBDA [X Y] [COND X [COND Y [LTE [CDR X] [CDR Y]] NIL] T]]]

[ADDTOLIS
 [LAMBDA
    [X Y]
    [COND Y
          [COND [LTE X [CAR Y]] [CONS X Y] [CONS [CAR Y] [ADDTOLIS X [CDR Y]]]]
          [CONS X NIL]]]]

[IMPLIES [LAMBDA [X Y] [COND X [COND Y T NIL] T]]]

[AND [LAMBDA [X Y] [COND X [COND Y T NIL] NIL]]]

GENERALIZATIONS:
GENRL1 = [SORT A]

PROFILE: [/ [A] , / E N R / E N R G S1 [GENRL1] , / E N R / E N R / E N R / E N
R / E N R [GENRL11 A1] , / E N R / E N R .]

TIME: 57.06 SECS.
```

Note that three inductions are done and two interesting lemmas are discovered: ADDTOLIS preserves ORDERED and LTE is a connex relation. The proof succeeds when the goal reduces to T above. The prover then lists the relevant function definitions and generalizations done. The profile is a debugging tool, describing the sequence of proof techniques used, e.g., Evaluation, Normalization, etc., and which variables were inducted upon. We could actually input a partial profile and stop the prover at that point to inspect the state.

We maintained a regression suite, called the “proveall,” and re-ran PLTP on the entire suite whenever we changed the code. This was in response to our experience with other provers (ours and others’): changes in the search strategy could give us bragging rights to some “new” theorem without us being aware that the prover could no longer prove an “old” one we were still bragging about!

5.4 The Code

The compiled POP-2 for the theorem prover occupied 15,000 24-bit words on the ICL 4130. The typical successful theorem in the proveall suite took 8-10 seconds of CPU time on a machine that took 400 microseconds to do a `cons`. The hardest theorems, those about `SORT`, took 40-150 seconds.

By the way, the name “PLTP” does not appear in our early papers or code. We tended to call it “the theorem prover” and when more precision was needed we called it “the Edinburgh Pure Lisp theorem prover.”

PLTP was described in my dissertation [42] in 1973. The dissertation also lists 47 definitions and 67 theorems in its proveall regression suite. Virtually all of the theorems had never been proved automatically before because virtually all of them require induction. PLTP was presented at IJCAI in 1973 and a longer version of that same paper was published as “Proving Theorems about LISP Functions” [6] in 1975.

When John McCarthy saw PLTP his comment to me was “What took so long?” It was 12 years after “A Basis for a Mathematical Theory of Computation” [41].

There is to my knowledge no working implementation of POP-2 today and so the original PLTP source code cannot be run. But Grant Passmore has reconstructed PLTP in OCAML from my dissertation and I have coded PLTP in ACL2 working from old line printer listings from 1973.

The PLTP Archive contains both of the reconstructions; it may be found at `http://www.cs.utexas.edu/users/moore/best-ideas/pltp/index.html`. Also there you’ll find a discussion of the bugs found during a close code inspection after 40 years of theorem proving experience. More importantly for historians, it includes scanned copies of all our line printer listings from 1973, including the original POP-2 source code for PLTP, the proveall files, and PLTP’s proof output. Also in the archive are copies of the POP-2 reference manuals for the ICL 4130.

Despite its simplicity, PLTP convinced us that Lisp was quite expressive as a logic, that the duality between recursion and induction could be exploited to automate the selection of frequently appropriate induction schemes, and that our heuristics for controlling the proof process to exploit induction were effective.

6 “Thm”

Boyer and I left Edinburgh in December, 1973, and returned to the United States (Boyer to SRI, me to Xerox PARC, though I joined him at SRI in 1976). We re-implemented PLTP in Interlisp and continued working on it. Having proved to ourselves that the basic approach was viable we were interested in tackling more practical verification problems. For example, I verified a binary addition algorithm with PLTP in 1975 [43] and by the late 1970s we would be trying to model the BDX930 flight control computer [15]. Such goals required a richer logic and better automation.

By 1979 PLTP had evolved into the the prover described in the book *A Computational Logic* [8]. We still called it "the theorem prover" but because it resided on a directory named `thm` I will call it "Thm" here.

I describe the changes in the logic and the prover below. No publications are cited because the logic, the prover, and its proveall regression suite are contained in the book cited above [8]. Indeed, within a decade of the publication of the book, Boyer and I received word from perhaps half-a-dozen people that they had re-implemented Thm from it and that it had passed its regression. Sadly, we have not posted a copy of the Interlisp code for Thm, though we have line printer listings of the code, the proveall regression suite, and the output on the proveall. The best reference for Thm is *A Computational Logic* [8].

6.1 Logic

Like PLTP's, Thm's logic is a homegrown version of Lisp formalized in an untyped first-order logic of total recursive functions with induction, but we changed it significantly to make it more like a conventional programming language – the idea that numbers were lists of `NIL`s was perfectly logical but jarring – with more flexible data types.

Thm has two distinct primitive constants, `(TRUE)` and `(FALSE)`, abbreviated `T` and `F`. `F` replaced PLTP's `NIL` in the role of "false." Otherwise, `IF`, `EQUAL` and the logical connectives are as in PLTP.

Thm allows the user to introduce new data types with the "Shell Principle" but the language remains untyped in the sense that well-formed formulas impose no restrictions on the arguments to functions.

There are three built-in applications of the Shell Principle to construct the natural numbers, the literal atoms or symbols (including `NIL`), and conses. Abbreviations are introduced to allow the conventional notation for constants. The user can invoke the Shell Principle to add other types. shell accessors or "destructors" like `SUB1`, `CAR` and `CDR`, are axiomatized by the Shell Principle to return some specified default value, e.g., `0` for numbers, `NIL` for lists, when presented with input of the "wrong" type.

The function `LESSP` ("less than") on the naturals is defined recursively but built into the system because it is known to be well-founded. Each shell introduces a new well-founded relation. Lexicographic combinations of well-founded relations are also known to be well-founded. We were aware that we could formalize the ordinals up to ϵ_0 but chose not to at that time [8], p. 181.

The Definitional Principle allows arbitrary recursion provided Thm can prove there is a measure of the formals and a well-founded relation such that the measure decreases in every recursive call. In contrast, PLTP has no definitional principle. PLTP just adds purported definitions as axioms at the user's risk of unsoundness. Thm's Definitional Principle guarantees conservative extension of the theory. Thm does not admit mutual recursion.

The Induction Principle allows inductions to be similarly justified by a measure and well-founded relation.

It is instructive to compare the most elementary inductions of PLTP and Thm. Consider the function `APPEND`, whose Thm definition is

```
(DEFUN APPEND (X Y)
  (IF (CONSP X)
      (CONS (CAR X) (APPEND (CDR X) Y))
      Y))
```

and whose PLTP definition is similar except for the idiom used to test whether `X` is empty. The induction scheme for `(`ψ` X Y)` suggested to PLTP by `(APPEND X Y)` is

```
(AND (ψ NIL)
     (IMPLIES (ψ X Y) (ψ (CONS X1 X) Y)))
```

but the induction suggested to Thm is

```
(AND (IMPLIES (NOT (CONSP X)) (ψ X Y))
     (IMPLIES (AND (CONSP X)
                   (ψ (CDR X) Y))
              (ψ X Y)))
```

The main advantage of the Thm-style scheme is that the induction hypotheses could be expressed in terms of "destructors" (like `CDR`) that need not just be structural components of the objects being inducted upon.

For example, using the Thm Definitional Principle we can define `(GCD X Y)` to recur by subtracting the smaller argument from the larger after proving that the result is smaller. Then an induction for `(GCD X Y)` can, under an appropriate case analysis, assume the conjecture for `(- X Y)` or `(- Y X)` as needed. PLTP cannot express this induction because the difference between the two variables is not a fixed structural component of either. Similarly, Thm can deal with functions like the applicative quick sort which recurs on computed lists that are not structural components, and with functions that count up towards some limit. Thm can also deal with functions that accumulate results into a "sink," which require inductive hypotheses in which some variables are replaced by arbitrarily bigger terms.

6.2 Prover

The Thm prover is obviously related to PLTP but considerably more sophisticated. Perhaps the most far reaching change from PLTP to Thm is the provision for user-supplied lemmas. But to explain the design we must first discuss other changes.

6.2.1 Clauses

We abandoned the simple idea of representing all the subgoals in a single `IF` expression and adopted clauses to represent the different branches of a goal.

Instead of tackling `(IF` p `(IF` q r s`)` u`)` as one goal, Thm breaks it into three clauses, implicitly conjoined, $\{$`(NOT` p`)` `(NOT` q`)` $r\}$, $\{$`(NOT` p`)` q $s\}$ and $\{p$ $u\}$, representing `(IMPLIES (AND` p q`)` r`)`, `(IMPLIES (AND` p `(NOT` q`))` s`)` and `(IMPLIES (NOT` p`)` u`)`. This allows the various proof techniques to work on different conjuncts independently.

All the goal clauses are kept in a "pool" and each must be proved for the overall proof to succeed. We generalized the control architecture of the prover into what we called "the waterfall." A clause is picked from the pool, undergoes the cascade of proof processes – which might split the clause into new subgoals by the introduction of `IF` terms, and each new subgoal is dropped into the pool to start the process over. If a clause survives unchanged through the cascade, induction is tried and the base cases and induction steps are dropped into the pool.

6.2.2 Type-sets

To support reasoning about the several data types allowed by the Shell Principle we introduced the notion of "type-set" which, abstractly, is the set of shells to which the value of a term might belong and which is represented concretely by a bit mask manipulated by logical operations. For example, when assuming `(EQUAL` a b`)` true Thm computes the type sets of a and b and intersects them to get some new type-set ts; if ts is empty, the equality cannot be true; otherwise Thm asserts that `(EQUAL` a b`)` had the type-set of `T` but also that both a and b have type-set ts.

6.2.3 Simplification

We combined PLTP's three processes of evaluation, normalization, and reduction into a single process called simplification that operates on clauses. Using axioms and rewrite rules derived from lemmas (below) the simplifier rewrites each literal of the clause in turn, assuming the negations of the other literals. When a literal rewrites to an expression containing an `IF`, the branches are split out and spliced into the clause, producing several clauses. Assumptions are stored in a "type alist" that pairs terms with their types expressed as type-sets.

6.2.4 Definition-time Analysis

To support both the Definitional Principle and the Induction Principle, and in a radical and important break from PLTP, we introduced extensive processing of new definitions. The processing identifies the formal parameters controlling the recursion (i.e., those measured) and builds an "induction machine" that succinctly describes the induction schema suggested by the function and its justification. This eliminates the “bomb list” and allows more sophisticated induction analysis and selection. Unlike even ACL2, Nqthm automatically considered all non-trivial lexicographic combinations of measures suggested by lemmas.

We refined our heuristics to take account of the controllers of a function so as not to be fooled by “sinks” or other formals that change in recursion but were not germane to termination. Expansion is allowed if the controllers get simpler or already occur in the formula, even if other arguments do not.

The induction heuristics were improved, taking account of controllers to more carefully control merging, introducing a notion that one scheme subsumes another and a notion that one scheme “flaws” another, and consideration of measures in final selection. These notions eliminate many occasions in which PLTP chooses arbitrarily between schemes that it considers equally plausible.

6.2.5 Destructor Elimination

We added a new proof process called “destructor elimination” that is tried after a clause is fully simplified.

For example, if `X` is known to be a cons and the conjecture mentions `(CAR X)` and/or `(CDR X)`, destructor elimination replaces `X` by `(CONS A B)`, for new variables `A` and `B`. Thus, `(CAR X)` becomes `A`, `(CDR X)` becomes `B` and other occurrences of `X` become `(CONS A B)` which is easier to exploit.

For the `CONS` shell, the logical justification of this replacement is just the axiom `(IMPLIES (CONSP X) (EQUAL (CONS (CAR X) (CDR X)) X))`: use the axiom to replace certain `X`s by `(CONS (CAR X) (CDR X))` and then generalize `(CAR X)` to `A` and `(CDR X)` to `B`.

This simple idea is very helpful in dealing with `QUOTIENT` and `REMAINDER` where the inductively proved lemma

```
(IMPLIES (AND (NUMBERP N)
              (NUMBERP M)
              (NOT (EQUAL M 0)))
         (EQUAL (+ (REMAINDER N M) (* M (QUOTIENT N M))) N))
```

can subsequently be used to transform (ψ `N (REMAINDER N M) (QUOTIENT N M))` into (ψ `(+ R (* M Q)) R Q)`. Other lemmas can be used to bound `R` below `M`.

6.2.6 Elimination of Irrelevance

We added a new proof process, just before induction, called “elimination of irrelevance.” Because clauses allow separate branches to be dealt with separately it is sometimes the case that clauses contain cliques of isolated falsifiable literals that can just be dropped, making subsequent induction choices more appropriate.

6.2.7 Lemmas

The most far-reaching change was the allowance for user-supplied lemmas. The idea is that the user should be able to "configure" the prover for a proof in a given domain by formulating lemmas that are used automatically, after the lemma is proved by the system. This kept the prover solely responsible for soundness while letting the user contribute creative insights.

When the user suggests a lemma to prove, he or she also tags it with any of four tags indicating whether the lemma is to be used as a rewrite rule, a destructor elimination rule, a generalization rule, or an induction rule.

Most lemmas are tagged as rewrite rules. A lemma of the form

(IMPLIES (AND h_1 ... h_k) (EQUAL *lhs* *rhs*))

tagged as a rewrite rule has the following operational meaning: If the simplifier encounters a target term matching *lhs* under the substitution σ (i.e., the target term is lhs/σ) and the hypotheses governing the occurrence of the target imply the h_i/σ, then replace the target by rhs/σ. Rewriting is used recursively to establish the h_i/σ.

If the conclusion is (IFF *lhs* *rhs*) the same behavior is implemented but only in argument positions for which propositional equivalence is sufficient. If the conclusion is any other term, it is treated as though it were a propositional equivalence to T.

This allows the user to "program" the simplifier and required us to invent various heuristics for stopping backchaining, looping rewrites, and dealing with "free variables" in the h_i, i.e., variables not bound in σ.

The other three tags are relatively simple: Destructor elimination rules are illustrated above. Generalization rules are used to add restrictions on new variables entering the conjecture by generalization or elimination. Induction rules (which should have been called "measure and well-founded relation rules") are searched when a new definition is admitted to try to find a measure and well-founded relation justifying the recursion (and thus the suggested) induction.

Because of lemmas, Thm is an "interactive automatic theorem prover" whose behavior is fully automatic once a proof begins, but whose behavior can be influenced by the user. It puts creative burdens on the user, usually exercised in response to a failed proof attempt by the system. "What fact do I know that it doesn't?" "How can the fact be expressed generally enough to be proved?" "How can the fact be expressed so as to be used effectively?" But it relieves the user of directing low-level inference steps. This started the development of what later became known (facetiously) as "The Method" for discovering lemmas that involves the user and the prover in a synergistic dialog in which neither party understands exactly what the other is doing.

6.3 Landmark Theorems

Appendix A of [8] lists 404 definitions and theorems in the regression suite. The suite includes basically all of the theorems in PLTP's regression, but now restated into Thm's logical setting.

The most significant "new" theorems are discussed in Chapters IV, XVII, XVIII, and XIX: the soundness and completeness of a propositional tautology checker, correctness of a simple optimizing expression compiler, the correctness of the Boyer-Moore fast string searching algorithm, and the uniqueness of prime factorizations.

All four of these examples are noteworthy.

The first begins to hint at the idea of using verified Lisp programs to extend the capabilities of the prover.

The compiler proof was our first foray into the formalization of the fetch-execute cycle of a state machine. The machine was thought of as simple calculator – it has no branch or conditional instructions – so a program is just a linear sequence of instructions comparable to the button-pushes a human might use to compute an arithmetic expression. But its definition involves a "big switch" to decode the instruction. Running a program generated by compiling a "large" expression was relatively slow since the only way to run our Pure Lisp was to simplify a ground term. One must also recall that even constants were represented as ground function applications. The number `3` really was represented internally as `(ADD1 (ADD1 (ADD1 (ZERO))))`. Trying to compute, say, 2^{32}, easily expressed as the ground term `(EXPT 2 32)`, was impractical.

The fast string searching proof was a demonstration that Thm, and Lisp as a specification language and logic, could be used with conventional program semantics methods. Boyer and I invented the string searching algorithm in 1975 (the submission date of "A Fast String Searching Algorithm" [7] is shown as June, 1975). For *A Computational Logic* we chose to verify a simplified version of that algorithm via the inductive assertion method. We generated verification conditions in the Thm logic (by hand) and proved them with Thm. The proofs, of course, involve induction because they deal with such "unfamiliar" numerical concepts as the position of one string (list of character codes) in another, the position of the last occurrence of a character in a string, the distance between two occurrences, etc. These concepts are "unfamiliar" in the sense that they are not likely to be built into a pre-existing arithmetic prover. But they are easily defined recursively, and the basic properties of these concepts require induction to prove.

As we wrote,

> Since mathematical induction is a fundamental rule of inference for the objects about which computer programmers think (e.g., integers, sequences, trees), it is surprising that anyone would implement a theorem-prover for program verification that could not make inductive arguments. — [8], p. *xii*.

Finally, the existence and uniqueness of prime factorizations was our deepest foray into elementary number theory at the time.

> The principal difficulty behind the proof is that Euclid's greatest common divisor function (`GCD`) plays an important role, even though it is not involved in the statement of the theorem. A beautiful but surprising fact (that multiplication distributes over `GCD`) is used; the more obvious fact that the `GCD` of two numbers divides both of them is also used. No other theorem yet proved by the theorem prover employs as a lemma a surprising fact about a function not involved in the statement of the theorem. — [8], p. 309

This example thus highlights two important features of the design of Thm. The first is that having a conservative Definitional Principle is crucial. If a formula does not involve a given function symbol but can be be proved after introducing the function with a conservative Definitional Principle, then the formula is a theorem of the original theory. Definitions, such as `GCD` and, for that matter, of the position of the last occurrence of c in str, are critical to inductive proofs of their respective goal theorems.[8]

The second important feature is the value of having a human in the loop. `GCD` and its properties were stated as lemmas by us, but verified by the prover and used (as we anticipated) in the deeper proof.

Thm's main positive contributions to our thinking were (a) the value of a Definitional Principle so closely coupled with an Induction Principle, (b) the effectiveness of definition-time analysis and data base organization, and, mainly, (c) the incredible power achieved by bringing a human into the loop without asking him or her to direct individual proof steps.

One negative observation we made of our prime factorization proof was how hard it is to prove facts about arithmetic if rewriting is the only available technique.

Our focus on trying to model languages and computing devices in the most realistic way possible exposed weaknesses of Thm, perhaps best summed up by Donald MacKenzie.

> In Boyer and Moore's emerging view, program verification, as it was practiced in the 1970s, had almost a theological quality in its abstraction from the machines on which programs ran. Dominant approaches, they suggested, implicitly assumed that it was a "god" rather than a physical machine, "that actually steps through one's programs." Boyer and Moore concluded that to verify SIFT [Software Implemented Fault Tolerance, a major SRI project], it was necessary to construct a mathematical model of what BDX930 machine code instructions did. They sought to encode in the logic used in their theorem prover a function that described the changes in the state of the BDX930 effected by each

[8] To demonstrate conservativity, Ken Kunen proved the prime factorization theorem without introducing `GCD`; see [13], Nqthm example file 52, p. 430.

instruction. The formula for the function covered thirty pages[9], even though it was described not in the most primitive terms possible but in terms of functions such as 8-bit (binary digit) addition. The formula was so large that it could not be processed by their theorem prover. — [39] p. 240

7 Nqthm

In 1981, Boyer and I left SRI and moved to the Computer Science Department of the University of Texas at Austin. We were located in the UT Tower in Don Good's verification group, and we started teaching for the first time. Of particular note was a graduate course of our invention called *Recursion and Induction* through which we met many amazing students, many of whom would become master-level users of our provers. We continued to evolve our prover, focused on more efficient representation of constants like 2^{32} or the assembly code program in some machine, support for arithmetic reasoning, and ways to extend the prover without having to change the code base.

Seven years after moving to UT we published *A Computational Logic Handbook* (first edition) [11] which described the prover that became known as Nqthm.[10] A second edition of the book came out in 1998 [13]. The Preface to the First Edition ([13], p. xv) contains an interesting recounting of what we thought in 1988 were the major changes from Thm to Nqthm. The list below is more complete and benefits from hindsight.

7.1 Logic

Nqthm's logic is a homegrown Lisp formalized in untyped, first-order logic of total recursive functions with induction quite similar to Thm's. But the differences are important. We introduced a new abbreviation convention to allow efficient representation of constants, formalized the ordinals below ϵ_0 to allow a wider variety of definitions and inductions, and axiomatized a partial recursive interpreter to allow bounded quantifiers and support reflection.

[9] See [15].

[10] Before ACL2, Boyer and I did not name our provers – or our other ideas like the "text editor" and "the fast string searching algorithm." We referred to whichever prover we were working on as "the theorem prover." This way of speaking made more sense when we were the only people using the tool. Of course, code resides on directories and the directories had names. In Edinburgh, there were no directories as such, just numbered disk tracks, and PLTP resided across several. When "the text editor" was used by others it was called by its track number, 77. But at PARC and SRI the prover resided on named directories and these tended to change as the code morphed. But at one point the directory was `thm` and that forked `qthm` to experiment with a quantified version, supporting bounded quantification. But we abandoned `qthm` and forked again, to `nqthm`, "new qthm," to experiment with the interpreter described below. Colleagues and students started calling the theorem prover "Nqthm."

7.1.1 A New Abbreviation Convention (and Term Representation)

We extended the abbreviation conventions with Lisp's `QUOTE` notation to make it possible to write arbitrary constants in the logic. In Thm, `3` is just an abbreviation for `(ADD1 (ADD1 (ADD1 (ZERO))))` and is represented that way as a formal term. But in Nqthm, `3` still abbreviates that same nest of `ADD1` terms but is internally represented as the quotation of Interlisp integer `3`, i.e., `(QUOTE 3)`, which, in Lisp, may be written `'3`. All constants were marked with `QUOTE`. The `LITATOM` (symbol) constant whose name is `ABC` is formally the term `(PACK (CONS 65 (CONS 66 (CONS 67 0))))`, where `PACK` is the constructor for the `LITATOM` shell, and 65, 66, and 67 are the ASCII codes for `A`, `B`, and `C`. But Nqthm represents that term as the quotation of the Interlisp symbol of the same name, `'ABC`. Similar remarks hold for list constants.

This allows Nqthm to represent relatively large constant objects efficiently, recognize them easily, and compute on them. But an important side-effect, indeed, the motivation behind this abbreviation convention and change in internal representation, was that every formal term in the logic had a corresponding constant in the logic obtained by writing the term inside a `QUOTE` expression. We call this formal object the "quotation" of the term. For example, the quotation of the term `(EQUAL (REV (REV X)) X)` is `'(EQUAL (REV (REV X)) X)`, which is a list of length 3 whose `CAR` is the `LITATOM 'EQUAL`, etc. From the perspective of the logic, this is not a change in the axioms or rules of inference, but just a new abbreviation convention for constants.

7.1.2 Ordinals

We formalized the ordinals up to $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$ following Gentzen's [22] treatment of Cantor normal form, except using lists and natural numbers.

Table 1 shows some ordinals and their representation in Nqthm.

`(ORDINALP X)` is recursively defined to be `T` or `F` depending on whether `X` is an ordinal, and `(ORD-LESSP X Y)` is defined to be `T` or `F` according to whether the ordinal `X` is smaller than the ordinal `Y`. `ORD-LESSP` was assumed to be well-founded.[11]

The Definitional and Induction Principles are like Thm's except they require the measure to return an `ORDINALP` and they replace Thm's limited selection of well-founded relations with `ORD-LESSP`.

Therefore, Nqthm can induct up to ϵ_0. As noted in [11] p. 42, "Gentzen's proof of the consistency of elementary number theory uses induction up to ϵ_0. Thus, by Gödel's incompleteness theorem, we know that induction up to ϵ_0 cannot be justified in elementary number theory."

[11] This assumption is backed by a careful hand-proof presented in a comment at the bottom of the source file `basis.lisp` in the Nqthm sources described below.

Ordinal	Nqthm object
0	`0`
1	`1`
2	`2`
...	...
ω	`(1 . 0)`
$\omega + 1$	`(1 . 1)`
$\omega + 2$	`(1 . 2)`
...	...
$\omega \times 2 = \omega + \omega$	`(1 1 . 0)`
$(\omega \times 2) + 1$	`(1 1 . 1)`
...	...
$\omega \times 3 = \omega + (\omega \times 2)$	`(1 1 1 . 0)`
$(\omega \times 3) + 1$	`(1 1 1 . 1)`
...	...
ω^2	`(2 . 0)`
...	...
$\omega^2 + (\omega \times 4) + 3$	`(2 1 1 1 1 . 3)`
...	...
ω^3	`(3 . 0)`
...	...
ω^ω	`((1 . 0) . 0)`
...	...
$\omega^\omega + \omega^{99} + (\omega \times 4) + 3$	`((1 . 0) 99 1 1 1 1 . 3)`
...	...
$\omega^{(\omega^2)}$	`((2 . 0) . 0)`
...	...
$\omega^{(\omega^\omega)}$	`(((1 . 0) . 0) . 0)`
...	...

Table 1 Some Ordinals in Nqthm

7.1.3 An Interpreter

We added an interpreter for partial recursive functions called `V&C$`. The name stood for "value and cost." `(V&C$ X A)` took the quotation of a term and an alist (list of pairs) binding symbols to values and either returned a pair consisting of the value of the term under the alist and the natural number "cost" of computing it or returned `F` indicating that no cost is sufficient. `V&C$` is itself a partial recursive function and thus an anomaly in Nqthm.

`EVAL$` is defined just to be the `CAR` of the value and cost, or `F`.

With `V&C$` Nqthm could allow the definition of partial recursive functions and allow proofs that some were non-terminating. More practically, `V&C$` allows the user to define bounded generalized quantifiers like the function `SIGMA` that sums the value of a (quoted) expression as a given (quoted) variable ranges over a given list. E.g., `(SIGMA 'I '(1 2 3) '(TIMES I I))` $= (1^2 + 2^2 + 3^2) = 14$.

But the main use of `V&C$` (actually, of `EVAL$`) was to support what is now called "reflection" and what Nqthm and ACL2 call "metafunctions": functions that operate on the quotations of terms and are proved to preserve the meaning (`EVAL$`) of the term, thus allowing the extension of the prover with verified simplifiers, etc. No change in the logic, beyond the provision of `EVAL$`, is necessary to justify the use of verified metafunctions in a first order setting.

The crucial property of `EVAL$` for these purposes is that for a certain class of terms and variable bindings, the `EVAL$` of the quotation of a term in the class is equal to the term. We discuss metafunctions further below when we describe changes to the prover.

Despite our pride in having come up with a way to add soundly a partial recursive universal interpreter to a first-order logic of otherwise total recursive functions, I will not present more details of `V&C$` here. For a detailed discussion of `V&C$` and some theorems about partial recursive functions and bounded quantifiers proved with it, see [3]. When we moved from Nqthm to ACL2 we abandoned `V&C$` in favor of a simpler way to justify metafunctions, one not requiring the admittance to the logic of a partial recursive universal interpreter.

7.2 Prover

The main improvements to Thm's proof techniques were the elaboration of the interpretation of rewrite rules, the integration of a linear arithmetic decision procedure, the support for metafunctions, provision for user-supplied explicit hints, a derived rule of inference called "functional instantiation," a Skolemizer providing first-order quantification, and support for faster execution on ground terms. All but the last were aimed at giving the user more control while maintaining (or, in the case of linear arithmetic, increasing) the degree of automation.

7.2.1 Rewrite Rules

To Thm, the operational meaning of a lemma tagged "rewrite" is always as a conditional rewrite rule. But Nqthm inspects the shape of the rule and gives certain shapes different meanings. One special shape is a rule that can be used to infer the type-set of a term, another shape identifies a predicate as just a conjunction or disjunction of types and improves the types computed when the predicate is assumed true or false by the simplifier, and the last special shape indicates the lemma can be used by the linear arithmetic decision procedure, which we discuss further below. All other shapes are used as conditional rewrite rules.

7.2.2 Linear Arithmetic

In the late 1970s, when Boyer and I were at still at SRI, we began considering ways to exploit decision procedures for linear inequalities. Our SRI colleague, Rob Shostak, was thinking about such decision procedures [53] and our nearby Stanford colleague, Greg Nelson, was thinking about combining decision procedures [47]. So this was "in the air." But our problem was different because we were trying to integrate a decision procedure into a heuristic prover for a much richer theory than just the arithmetic of addition, subtraction, constants, variables, and inequality relations.

As was made clear in our work on the prime factorization theorem, handling all arithmetic solely with conditional rewrite rules is tedious. Consider the transitivity of less-than (in ACL2 notation).

```
(IMPLIES (AND (< X Y)
              (< Y Z))
         (< X Z))
```

This is a fundamental tool in arithmetic and yet when used as a conditional rewrite rule it means "if you're trying to prove an instance of `(< X Z)` – i.e., any inequality – find a `Y` that allows you to prove `(< X Y)` and `(< Y Z)`. This gives rise to an enormous search space, unless the "find a `Y`" step is limited to, say, terms already in the formula, which weakens the utility of transitivity and requires the user to formulate *ad hoc* lemmas to suggest the right `Y`.

But, as we wrote in the 1988 paper describing the integration

> We discuss the problem of incorporating into a heuristic theorem prover a decision procedure for a fragment of the logic. An obvious goal when incorporating such a procedure is to reduce the search space explored by the heuristic component of the system, as would be achieved by eliminating from the systems's data base some explicitly stated axioms. For example, if a decision procedure for linear inequalities is added, one would hope to eliminate the explicit consideration of the transitivity axioms. However, the decision procedure must then be used in all the ways the eliminated axioms might have been. The difficulty of achieving this degree of integration is more dependent upon the complexity of the heuristic component than upon that of the decision procedure. The view of the decision procedure as a 'black box' is frequently destroyed by the need to pass large amounts of search strategic information back and forth between the two components. Finally, the efficiency of the decision procedure may be virtually irrelevant; the efficiency of the final system may depend most heavily on how easy it is to communicate between the two components. — [12] p. 83

It took us four years of experimenting before we settled on the final integration. We selected a rather old fashioned decision procedure for linear inequalities over the rationals [27] based on cross-multiplying and adding inequalities to eliminate variables. We gave up completeness on the integers since one can use induction when that is a problem. The decision procedure is relatively inefficient but operates on formulas rather than, say, graphs, so that communication between the prover and the procedure does not require a radical change of representation.

Communication happens in both directions: when the rewriter needs to establish an inequality it uses the linear procedure, and when the linear procedure needs to relieve the hypotheses of a "linear lemma" it uses the rewriter. And example linear lemma might say that `(LASTPOS CHAR STR)`, which is just a "variable" in the linear procedure, is less than (LENGTH STR), if `STR` is a string and `CHAR` occurs anywhere in it. The linear procedure calls the rewriter to establish the hypotheses.

The lemma suggested above is an example of a lemma that if tagged with the rewrite tag is actually built into the linear procedure.

Our paper on how to integrate a decision procedure into a heuristic prover [12] recounts the various approaches we tried and how we were gradually driven to a tight integration. It took several more years to find a venue for that paper because it did not introduce a new decision procedure but I have been told by several theorem proving researchers that it was helpful. The experience also led us to write

> The difference between a black box and an integrated decision procedure is a lot of work. It is probably the case that much hard work on any given black box will be scrapped when the box is torn apart and reassembled inside a larger system. Indeed, we believe that the work on many procedures is simply irrelevant to the goal of constructing useful mechanical theorem provers since the use of a faster procedure will not necessarily speed up the overall system. We believe that the development of useful procedures for program verification must take into consideration the problems of connecting those procedures to more powerful theorem provers. — [12] p. 123

I still hold this position, but I am encouraged by the modern work on combining decision procedures, SMT, predicate abstraction, and other methods of exploiting decision procedures. Furthermore, it must be said that the integration of linear arithmetic, hard though it was, was worth it because it reduces the burden on the user and opens the door to much more interesting arithmetic proofs.

7.2.3 Metafunctions

"Metafunction" was our name for a function that takes as input the quotation of a term and is verified to return the quotation of an equivalent term. The relation between the input and output is that their `EVAL$`s are the same. When a theorem of that shape is proved and has been tagged `META`, Nqthm builds the function into its rewriter.

The first metafunction we defined and verified was named `CANCEL`. It operated on (the quotations of) terms of the form `(EQUAL lhs rhs)`. The idea was to cancel common addends on either side of the equation. This is hard to do with rewrite rules. For example, the rewrite rule concluding with

```
(IFF (EQUAL (+ X Y) (+ X Z))
     (EQUAL Y Z))
```

could not be used to cancel the `A2` from both sides of `(EQUAL (+ A1 (+ A2 B)) (+ A2 (+ C D)))` and no fixed set of our rules would handle all possible combinations.

But CANCEL worked on (the quotation of) (EQUAL lhs rhs) by computing the list of addends in lhs and rhs, intersecting them, deleting the intersection from each list, reassembling +-expressions from the new lists and embedding them in an EQUAL expression.

Given that CANCEL is proved correct, meaning that the EVAL$ of (CANCEL X) is the EVAL$ of X, and the theorem that a term is equal to the EVAL$ of its quotation (under a suitable alist), the use of CANCEL to replace, say (EQUAL (+ A1 (+ A2 B)) (+ A2 (+ C D))) by (EQUAL (+ A1 B) (+ C D)) requires no further meta-logical reasoning:

```
(EQUAL (+ A1 (+ A2 B)) (+ A2 (+ C D)))
= {EVAL$ of quotations}
(EVAL$ '(EQUAL (+ A1 (+ A2 B)) (+ A2 (+ C D))) (LIST (CONS 'A1 A1)
...)))
= {CANCEL correct}
(EVAL$ (CANCEL '(EQUAL (+ A1 (+ A2 B)) (+ A2 (+ C D))))
       (LIST (CONS 'A1 A1) ...))
= {def CANCEL}
(EVAL$ '(EQUAL (+ A1 B) (+ C D)) (LIST (CONS 'A1 A1) ...))
= {EVAL$ of quotations}
(EQUAL (+ A1 B) (+ C D))
```

Furthermore, it is very fast and efficient provided functions in the logic, like CANCEL, can be computed efficiently. The speed with which Nqthm can go from a term to its quotation (basically a no-op because they are both represented by the same Lisp object except for the QUOTE) is one key to the efficiency of metafunctions. Indeed, it is impractical to represent the quotation of a typical term as an object in Thm's ground-term representation.

Metafunctions are quite useful in proofs about models of von Neumann computing machines because much time is spent in normalizing the arithmetic expressions denoting addresses and comparing such normalized addresses to see if a "write" interferes with a "read." They became a staple of the Nqthm and, especially, the ACL2 repertoire of proof techniques.

Our first metafunction, CANCEL, was proved correct by Thm in October, 1978, long before Thm had the machinery to use it. The original metafunction paper (SRI CSL Memo 180, December, 1979) was the basis of a 30-hour seminar series as part of IBM Belgium's establishment of the *chaire internationale d'informatique* in Liege, Belgium, in February, 1980 [9]. As presented at Liege, EVAL$ was not involved; we used a simpler function called MEANING. EVAL$ and V&C$ were not introduced until the mid-1980s [3].

The closest work related to metafunctions contemporary with ours is that of Weyhrauch [58]. However, his notion of reflection requires formalizing the proof theory to state the correctness. Furthermore, to get efficiency Weyhrauch suggests the perilous act of unverified "semantic attachment" of programs to "metafunctions."

7.2.4 Explicit Hints

Nqthm allows the user to provide hints to the prover. Hints specify the subgoal to which they are to be applied and are acted upon when that subgoal arises. Nqthm's hints are

– `USE`: add to the hypotheses of the subgoal specified instances of specified lemmas, basically implementing the advice "this subgoal follows by these (usually creative) instances of already proved facts"
– `EXPAND`: forcibly expand specified function calls, overriding the system's heuristics
– `ENABLE` and `DISABLE`: turn on or off specified lemmas; disabled lemmas are not automatically applied, allowing the user to enforce or break abstraction boundaries
– `HANDS-OFF`: do not rewrite these terms
– `INDUCT`: induct as suggested by this term, rather than as suggested by the goal
– `DO-NOT-INDUCT`: do not use induction, i.e., fail if the prover does not prove this goal without induction
– `DO-NOT-GENERALIZE`: do not generalize common subterms

In August, 1986, Matt Kaufmann joined the UT verification group. At the time his background included research in mathematical logic and programming languages. Within about a year he implemented a proof checker interface for Nqthm called PC-Nqthm [32]. While not technically a "hint" it allowed interactive proof checking via a goal manager, akin to those in LCF-style provers.

7.2.5 Functional Instantiation

We added the ability to introduce an undefined function symbol that is constrained to have certain properties, provided one proves that there exists a function with those properties. E.g., one might introduce an undefined relation known to be connex. One might then define a sort routine that uses this relation and prove that it produces an ordered permutation of its input. But then one might then want to instantiate the sort routine and its correctness theorem to get a function that sorts using alphabetic ordering. Functional instantiation [17] is a derived rule of inference that lets the user do this, provided (roughly speaking) the concrete functions substituted for the constrained ones satisfy the constraints on the undefined function. The facility is described in second edition of the Handbook [13].

The provision for constrained functions and the ability to exploit lemmas about them without having to re-construct every step, is very useful because in modeling complex machines one often finds state components that are modified in ways that are unimportant. With functional instantiation the user can model a "generic" machine, prove it correct, and then derive a concrete instance of it that can be executed and is known to be correct.

7.2.6 Defn-sk

In 1992, Kaufmann developed a Skolemizer for Nqthm [33] to support first order quantification without changing the syntax of Nqthm. `Defn-sk` allows the user to introduce of a function whose body contains outermost universal or existential quantifiers. It is axiomatized so that, in the case of an existential quantifier, if the function returns true then a "witness" function introduced with it returns an object satisfying the body of the quantified expression. The universal case is symmetric. Of course, these functions cannot be executed but their introductions are conservative. We included `defn-sk` in the second edition of the Handbook.

7.2.7 Executable Counterparts aka `*1*` Functions

Evaluation of ground terms was increasingly important in Nqthm because

- Ground terms arise in proofs, for example, when establishing that a certain initial machine state can be obtained by running the machine model a certain number of steps.
- Ground terms are used to test models and conjectures before trying to prove properties or to let the user find a counterexample to a formula he or she mistakenly thinks is a theorem but is having trouble proving.
- Ground terms arise every time a metafunction is applied.

To support the faster execution of functions in the logic, we compiled a version of each Nqthm function (including those introduced by the user). The problem is that some Nqthm functions behave differently than their Lisp counterparts. For example, while `(CAR 7)` in the Nqthm logic is `NIL`, `(CAR 7)` in Interlisp causes an error. So Nqthm includes a Lisp definition of a new function, `*1*CAR`, that agrees with the axioms. `(*1*CAR 7)` is `NIL`.

When the user defines `FOO`, Nqthm introduces an Interlisp function with the name `*1*FOO`. A logical term with function symbol `FOO` in which all the arguments are `QUOTE`d can be reduced to the constant described by the Nqthm axioms by running `*1*FOO` in the underlying Interlisp.

The generation of these `*1*` definitions was straightforward: essentially we just renamed every function symbol appearing the logical definition to its `*1*` counterpart. However, the `*1*` of shell accessors like `SUB1`, `CAR`, and `CDR` were defined in the Nqthm sources to test at runtime the types of the arguments and default them to the values used in the logic.

7.3 Landmark Theorems

An annotated bibliography of interesting theorems proved with Nqthm may be found on pages 5–13 of the second edition of *A Computational Logic Handbook* [13]. Proper citations are provided there. Among the important "new" theorems proved with Nqthm were the invertibility of the RSA encryption algorithm, the unsolvability of the halting problem for Pure Lisp, a variety of theorems about FORTRAN 77 programs verified with a FORTRAN 77 verification condition generator [10], the correctness of the FM8501, FM8502, and FM9001 microprocessor designs (Warren Hunt), the correctness of the Piton assembler/linker/loader that connected Hunt's microprocessors to higher level software in the CLI Verified Stack (Bill Bevier, Art Flatau, Warren Hunt, Matt Kaufmann, J Moore, Bill Young, Matt Wilding), Godel's incompleteness theorem (Shankar), Gauss' law of quadratic reciprocity ("the jewel of elementary number theory") (David Russinoff), Wilson's theorem (David Russinoff), the bi-phase mark asynchronous communications protocol, the verification of the Berkeley C String Library on an Nqthm model of the Motorola MC68020 (Yuan Yu), and the Paris-Harrington Ramsey Theorem (Ken Kunen).

Of particular note for the next evolutionary step in the development of the prover are the theorems about machine models. All of our models are operational ones, describing how to transform an input state to an output state, where states are just lists or other shell structures full of numbers or symbolic instructions and data. The FM9001 was a gate level description of a simple but full-function 32-bit microprocessor that was actually fabricated. The CLI Verified Stack was a hardware/software stack on FM9001 that included an assembler/linker/loader, two compilers, a simple operating system, and some application programs, all of which were proved correct with Nqthm.[12] Furthermore, the correctness theorems were logically composed. For example, Nqthm proved that a simple game playing program (written and proved with respect to the semantics of its high-level language) never loses. If that program is then compiled, assembled, linked, and loaded as a binary image on the FM9001 and then run, the FM9001 would never lose. Finally, consider the model of the MC68020. It consists of about 600 Nqthm functions spanning about 80 pages. An example program proved correct is a slightly modified version of the C code for Hoare's *in situ* quick sort. It was proved correct by using the Gnu C compiler, `gcc`, to produce binary code for the MC68020 and verifying the execution of that code.

Now contemplate trying to do that in Thm with its purely logical representation of constants. Recall the BDX930 project! In fact, contemplate the demands of simply running these logical functions on logical objects representing their binary images. However, despite Nqthm's success at managing these proofs, these projects were pushing it to the limits of its capacity.

7.4 Code

When we arrived at UT, the theorem prover we brought with us was coded in Interlisp. Of course, by the time we arrived at UT, Thm had already begun to morph into Nqthm with the addition of metafunctions (and the concomitant

[12] Actually, the Stack was first built and verified for Hunt's earlier FM8502, which was never fabricated. When Hunt and Brock verified the FM9001, which was fabricated, the assembler/linker/loader for the earlier stack was re-targeted to FM9001 and verified; the rest of the stack then built without change.

`QUOTE` notation) and the beginnings of experiments with linear arithmetic.[13] In 1983-84, we translated it into Maclisp for a DEC-2060 and a Honeywell Multics, and later into Zetalisp for a Symbolics 3600. Eventually, we re-coded it again into Common Lisp.

The Common Lisp source code for Nqthm is available at `https://www.cs.utexas.edu/ftp/boyer/nqthm/` along with its extensive regression suite that includes the landmark theorems. See [13], p. 423, for an annotated list of the examples in the regression suite.

A comment in the source file `basis.lisp` reads "As of February, 1984, this system contained about 15,500 lines of which about 2000 were comments and 1700 were blank – leaving about 12,000 lines of 'code'. There were about 550,000 characters in toto, which compiles into about 400,000 bytes of compiled code in Maclisp for the 2060 but only about 260,000 bytes in Zetalisp on a 3600."

8 ACL2

Before continuing with the technical details it is important to note a change of scene in our story. Between 1985 and 1989, the verification group in the UT Tower decided to move off campus and start a company whose mission was to spread verification technology. That company was Computational Logic, Inc. or CLI. The entire group moved as UT research contracts expired and CLI acquired new contracts. Our business plan was simply to contract our time and Nqthm expertise to verify hardware and software for government and industry. Along the way, of course, we would continue to develop the tool and distribute it without cost. Boyer and I were, at first, on leave without pay from UT to work at CLI, but in 1989 I resigned my professorship to devote full time to my projects at CLI. Boyer returned to UT but consulted at CLI. We both hired and supervised graduate students from UT to do their research at CLI.

As noted above, Nqthm was being used on problems vastly bigger than Boyer and I had imagined back in Edinburgh. Nqthm functions ran too slowly to allow extensive testing of big models, especially compared to the execution speeds available in well-declared Common Lisp. To take advantage of that speed, we decided to adopt an applicative subset of Common Lisp [55], a modern dialect of Lisp that would soon have an ANSI standard, as our logic. In August, 1989, we stopped developing Nqthm and started a completely new prover. We named that prover ACL2 for *A C*omputational *L*ogic for *A*pplicative *C*ommon *L*isp. We axiomatized a significant applicative subset of Common Lisp and proceeded to "re-code" Nqthm for that logic. Furthermore we confined our implementation language to the new logic itself as much

[13] The Interlisp source code and regression suite of an intermediate prover between Thm and Nqthm, is available at `https://drive.google.com/open?id=0B2yFYLn0Spf1WHFXVDRfZkk1Ukk`. It is dated 19 July, 1982. Unlike Thm, it contains some linear arithmetic support and metafunctions, but not `V&C`.

as possible. Several times we had to extend the subset of Common Lisp because of features we needed in the prover.

A comment in the ACL2 source code reads "Matt Kaufmann officially joined the project in August, 1993. He had previously generated a large number of comments, engaged in a number of design discussions, and written some code. Bob Boyer requested that he be removed as a co-author of ACL2 in April, 1995, because, in his view, he has worked so much less on the project in the last few years than Kaufmann and Moore." Matt and I have developed and maintained ACL2 for the past 25 years.[14]

Since this paper is about the technical evolution from PLTP to the industrially useful ACL2, I will not recount further the CLI experience except to say that is was without doubt the most pleasant place I have ever worked. The dedication of the people there was instrumental in the industrial success of ACL2. The paper "Industrial Hardware and Software Verification with ACL2" [56] tells how ACL2 became industrially relevant and the story is as much one of "soft power," i.e., passion and dedication in a diverse group of people, as of the technical power of the tool. To encourage the reader who wants a complete picture, I simply list the sections of that paper "The Origin Story,", "Initial Industrial Demonstrations," "Dispersion" (of the technical community to various companies using ACL2), "Integrating ACL2 into the Microprocessor Design Workflow," "ACL2 at Centaur Today" (a case study of one company), "Other Ongoing Industrial Projects," "Strengths and Weaknesses of ACL2 from an Industrial Perspective," "Prover Development and Maintenance," and "Lessons." If I had unlimited space here and copyright ownership of [56] I would include that paper verbatim!

Before discussing the evolution from Nqthm to ACL2 several points must be emphasized.

First, comprehensive user-level documentation for ACL2 is available online. I will take advantage of that in describing ACL2 and how it changed from Nqthm. When I write "[:DOC x]" here I am referring to the online documentation. Go to the ACL2 home page [31] in your browser, click on The User's Manuals link, then click on the ACL2+Books Manual link and type x into the **Jump to** box.

Second, despite its age (30 years have elapsed since we started it) it is still "living" code; it is changing. Forty two older versions (dating back to Version 1.9 released in Fall, 1996) are available in source code form at the ACL2 home page. Complete self-contained documentation for a release is part of each release. The documentation noted above is for the current release, Version 8.1, released in September, 2018.

Third, while the documentation for each release is stand-alone, each release has a note explaining the major changes since the previous release. See [:DOC release-notes]. For Version 8.1 that note is [:DOC note-8-1]. It lists about 80

[14] Kaufmann left CLI in August, 1995, and subsequently worked at Motorola, EDS, and AMD where he gained much experience with hardware design and simulation-based verification but continued to be a co-author of ACL2. He rejoined the UT Computer Science department in 2005.

changes ranging from fixes to two soundness bugs in Version 8.0 to new proof techniques and features. It is simply impossible – and would be redundant with available online resources – to list the thousands of changes to ACL2 made in the past 30 years.

Fourth, some important features of ACL2 are in certified files written by users. In ACL2, a "book" is a certified file of definitions, theorems, and other *events* extending the logic and configuring the prover. The ACL2 Community Books reside on GitHub[15] and are contributed, documented, and maintained by the ACL2 user community. As of January, 2019, there were 6,949 books in the repository containing over a hundred thousand theorems across a wide variety of domains. The Community Books also constitute the ACL2 regression suite. Before any release the new version must be able to successfully process every book in the repository. The online documentation mentioned above also includes the documentation of many books, thanks to tools written by Jared Davis.

Fifth, a huge amount of work has been devoted to engineering ACL2 for speed and capacity. For example, extending the bit-masks used for type-sets to represent the fact that the typed term might return `1` took many hours of work, but was of huge benefit to industrial users concerned with gate-level models. Similarly, months were spent arranging for ACL2 to load books faster by loading their previously compiled files first, but this cannot be done naively, e.g., ACL2 will reject the subsequent logical definition if it finds the name has a compiled definition in Lisp. Such changes are not contributions to theorem proving, but are absolutely necessary to produce a prover of use to industry. I do not focus on them below.

Finally, despite its size, ACL2's regression suite does not include many of the landmark theorems cited for Nqthm, theorems like Gödel's incompleteness theorem and the CLI Verified Stack, that I wish I could list in ACL2's repertoire. But ACL2's logic is not just a minor modification of Nqthm's and the heuristics in the provers are often different. This raises an obvious challenge to our claims of ACL2's power and flexibility: why not build an ACL2 book that makes ACL2 into Nqthm, able to process all the files in Nqthm's regression? In fact, I spent several months trying to do that with an early version of ACL2 and gave up; it is one thing to design an ACL2 script to prove a theorem proved by another prover but it is quite a different thing to configure a prover to follow another prover's script to prove a theorem stated in a different logic. But mainly I stopped because the ACL2 community had industrial challenges to face. Perhaps when I don't have much else to do I'll reconsider this project.

[15] The top of the directory structure is `https://github.com/acl2/acl2`. A particular file (book) may be found by clicking your way down the directory hierarchy. For example, to find `books/projects/codewalker/` start on the GitHub page above and click on `books`, then `projects`, etc.

8.1 Logic

The syntax of Common Lisp is much richer than that of Nqthm. For example, Common Lisp provides a powerful abbreviation facility in macros, the definition of "symbol packages" and the selection of a "current package," functions that return "multiple values," etc. ACL2 supports these embellishments. Symbol packages fundamentally changed the axiomatization of symbols; it was quite challenging to axiomatize a useful applicative version of packages, but they give users namespace separation with limited, controlled sharing. Another surprisingly difficult task was the logical implementation of Common Lisp's multiple value return mechanism that was consistent with the axioms, portable across many implementations of Common Lisp, and suitably efficient. But efficient support for multiple value returns is one of the reasons ACL2 performs well; it allows an ACL2 function to compute several different results in one sweep without using the heap.

ACL2's logic is an untyped, first order logic of total recursive functions. We axiomatized a large part of the applicative subset of Common Lisp as described in *Common Lisp the Language, Second Edition* [55] and then extended it with some functions that make applicative programming in the language more efficient while preserving the axiomatic semantics.

ACL2 supports five broad data types: character objects, strings, symbols, numbers, and pairs (hence lists and trees). It does not support Nqthm's shells. The symbol `NIL` plays the role of the false truth value as well as the empty list. The ACL2 numbers include positive and negative integers, rationals, and complex numbers with rational components. ACL2 does not provide the floating point numbers of Common Lisp, but there are many ACL2 books that do.[16]

Common Lisp is syntactically untyped and so is ACL2. However, like its predecessors, ACL2's axioms "complete" the primitives on inappropriate arguments. For example, `(CAR 7)` in ACL2 is axiomatized to be `NIL` but causes a runtime error in Common Lisp. This discrepancy between ACL2 and Common Lisp complicates the question of whether an ACL2 function runs correctly in Common Lisp. We address this when we discuss the prover.

About 200 Common Lisp primitives are axiomatized.

We then extend this subset of Common Lisp with some "new" functions (i.e., ones not in Common Lisp) to facilitate programming and support theorem proving.

– a single-threaded `state` object giving the user access to the file system and other system resources; by "single-threaded" we mean that any function that changes a field in the object must return the object, thus allowing updates to be implemented destructively. [:DOC state].
– a facility allowing the user to introduce other single-threaded objects together with a facility to set up "abstract" views of such objects. [:DOC

[16] E.g., [:DOC rtl]. An extension of ACL2 by Ruben Gamboa called ACL2(r) supports the reals via non-standard analysis [:DOC real].

stobj, defstobj, defabsstobj], allowing the maintenance of complicated invariants.
- input/output facilities [:DOC io]
- applicative property lists [:DOC getprop, putprop]
- applicative arrays, logically modeled as lists but backed by Common Lisp arrays [:DOC arrays]
- theory scoping, allowing one to export definitions and theorems from certain temporary conservative extensions of the current theory. [:DOC encapsulate, local].
- `defchoose`, a Skolem witnessing function, allowing the definition of `defun-sk` as a macro to provide first-order quantification within the quantifier-free syntax of ACL2. [:DOC defchoose, defun-sk].
- an "evaluation theory" which is an extension of the theory in which proofs are done. Ground terms typed at the top-level of the ACL2 read-eval-print loop are evaluated in the evaluation theory. The evaluation theory can be extended by attaching concrete functions to constrained ones, provided the concrete functions satisfy the appropriate constraints. This allows models containing constrained functions to be run. [:DOC defattach].
- limited second-order functionality allowing certain symbols and lambda expressions to be passed as "functions" and applied. This in turn supports a subset of the Common Lisp `loop` iterative construct [35].

A total of 1,168 functions are axiomatized in the initial state of ACL2 Version 8.1. A total of 452 functions, macros, and constants are documented in [:DOC ACL2-built-ins] but there are many other logically axiomatized functions that serve as "helpers," system functions, etc. Obviously, there is a lot more functionality than summarized here.

The Nqthm function `V&C$` was not included in ACL2. We recognized that we could justify metafunctions in terms of functional instantiation (again a derived rule of inference) and that with `defun-sk` we could introduce machinery to conveniently carry out some tasks formerly done with `V&C$`.

Like Nqthm, ACL2 formalizes the ordinals up to ϵ_0 (but with a more efficient representation [40]) and the Definitional and Induction Principles are the same. In addition, ACL2 allows mutually recursive definitions but, even though they are logically justified by well-founded relations as for singly recursive functions, calls of mutually recursive functions do not suggest inductions to the prover.[17]

8.2 Prover

In the early 1990s the ACL2 prover was essentially just a reimplementation of Nqthm's. But much additional functionality was added over time, including

[17] A user-written Community Book provides convenient support for proving inductive theorems about mutually recursive functions [:DOC make-flag].

- certified books allow the user to configure the system with definitions and lemmas (and thus rules) imported from different projects, including those from the ACL2 Community Books repository [:DOC books]. This is in contrast to Nqthm where every proof script has to list and prove every lemma.
- lists of rule names, called "theories," allow the user to enable and disable collections of rules [:DOC theory-functions, theory-invariant, theory-management].
- rewriting can be done with user-defined equivalence and congruence relations, not just equality and propositional equivalence [:DOC equivalence, congruence, refinement].[18]
- the linear arithmetic procedure is extended from just the naturals to the integers and rationals, and can also deal with some non-linear terms, i.e., products, although of course this is just heuristic [:DOC non-linear]. Robert Krug was instrumental in prototyping support for non-linear.
- to avoid more expensive rewriting and backchaining, a preprocessor is in the waterfall, before simplification, to catch "simple" cases like tautologies, subsumption by certain previously proved lemmas, theorems about "type-like" monadic predicates, and consequences of limited linear arithmetic [:DOC built-in-clause, introduction-to-the-tau-system].
- metafunctions are improved in a number of ways. They can compute hypotheses governing their correctness, hypotheses that must be proved upon application of the metafunction [:DOC meta]. They can be made faster by proving well-formedness of their output [:DOC well-formedness-guarantee]. They can be more powerful by accessing the theorem prover's internal state and the context of the term being rewritten [:DOC meta-extract].
- a facility akin to metafunctions, called "clause processors" is available allowing the user to write and verify functions that operate at the clausal level instead of the term level and can essentially serve as new processes in the waterfall [:DOC clause-processor].
- conditional rewrite rules can be better controlled. One facility allows the user to attach a syntactic predicate or "pragma" to a rule that must approve any application [:DOC syntaxp]. Another facility allows a hypothesis to be marked as "forced" meaning that it may be assumed true and if the entire proof succeeds then the full power of the theorem prover (instead of just rewriting) is brought to bear on the forced hypotheses, [:DOC force, case-split].
- forward chaining is available, allowing the assumptions available during rewriting to be extended [:DOC forward-chaining].
- several new ways of interpreting previously proved lemmas are available, giving the user better control at building in new information, [:DOC rule-classes].

[18] The original work on automated equivalential rewriting in this setting was done in 1988 as an experimental patch to Nqthm by Bishop Brock, then a student in Bob Boyers theorem proving class. The work was implemented in ACL2 in the early 1990s and has been generalized and improved over the years by Kaufmann and me.

- many new hints are available [:DOC hints]. One particularly powerful new facility allows the user to provide a function that computes an appropriate hint for a goal [:DOC computed-hints]. A facility within computed hints allows a hint to introduce a disjunctive split so that the full power of the prover is brought to bear on each of several ways to prove a clause. Another hint allows one to prove a subgoal with explicit low-level commands formulated in an interactive *proof-builder* for ACL2 that gives the user complete control with both low level logical inference steps and ACL2 tools like the simplifier [:DOC proof-builder].
- many tools are available for helping "debug" both definitions and failed proof attempts. Critical subgoals of failed proofs, called checkpoints, are displayed so the user focuses on [:DOC the-method]; most other output can be shut off or abbreviated [:DOC set-gag-mode]. Rewrite and other rules can be monitored and can cause interactive breaks when they are used [:DOC monitor, forward-chaining-reports]. [:DOC accumulated-persistence] indicates which rules are being heavily used and whether they are productive. Functions can be traced [:DOC trace]. Large formulae can be abbreviated [:DOC evisc-tuple, set-iprint].
- input to ACL2 is restricted to certain predefined forms such as definitions and theorems. However, ACL2 users often generate and submit such forms programmatically based on the current environment. A utility, `make-event`, allows the user to define new forms, taking advantage of ACL2's computational power. See [:DOC make-event].

8.3 Programming

One reason ACL2 has been successful in industry is that it is an efficient programming language (that can be verified with powerful mechanical aids). The idea that ACL2 is both a logic and a programming language imposes some requirements not imposed on Common Lisp. That has caused the addition of many features for either speeding up execution or making proofs simpler.

The main such features are the formalization of Common Lisp's implicit type system and support for Common Lisp's `DECLARE` facility. Declarations allow the Common Lisp compilers to generate extraordinarily efficient code. In the logical setting of ACL2, declarations generate proof obligations that must be dispatched to eliminate runtime tests. We actually spent years integrating `DECLARE` into ACL2.

In the spirit of Nqthm, ACL2 completes the Common Lisp standard [55] by defining functions where the standard leaves them unspecified. Axiomatically, `(CAR 7)` is `NIL` even though it causes unspecified errors in Common Lisp. Similarly, in ACL2, `EQ` (Lisp's pointer identity) is defined to be `EQUAL` (structural equality). This makes proofs about these functions uncomplicated but means they have properties beyond those specified in the standard. By generating "`*1*` functions" comparable to Nqthm's, ACL2 functions can be run according to the axioms. But `CAR` and `EQ` carry "guards" formalizing conditions that ensure that raw Common Lisp behaves as described by our axioms. `CAR` must be applied to a cons pair or to `NIL`, and at least one argument to `EQ` must be a symbol. If ACL2 proves the guard conjectures of a definition[19] then runtime errors cannot arise. To evaluate a call of a guard-verified function, ACL2 can avoid the slower `*1*` version and run the unfettered Common Lisp version just as the user typed it, including any declarations. This gives the user access to compiler optimizations, at the price of proof obligations above and beyond functional correctness. This can increase the speed by a factor of 10 or more.

ACL2 also provides a number of other ways to improve execution efficiency including a way to provide an easy-to-reason-about logical definition and an efficiently executable one – provided the two can be proved equivalent [:DOC mbe]. Functions can be memoized so that prior computations are not repeated [:DOC memoize]. Profiling facilities are provided to allow the user to find out where a formal model is spending its time [:DOC profile].

A major contribution was made by Bob Boyer and Warren Hunt in the early 2000s when they built a prototype of ACL2 providing "hash cons," which uniquely represents cons trees by sharing structure with previously allocated trees [:DOC hons] [16]. It is upon hash cons that memoization and applicative hash tables [:DOC fast-alists] are built and that in turn has allowed many applications to have the speed and capacity to be useful. After years of testing and use at Centaur, with code improvements by Jared Davis and Sol Swords, Boyer and Hunt turned their hash cons implementation over to Kaufmann and me for incorporation into the official sources.

Another source of ACL2's flexibility in building verification tools is the notion of "trust tags." Using trust tags [:DOC defttag] a user – generally a "super user" – can take responsibility for ACL2's soundness and modify or extend the source code in unverified ways. These changes can be packaged into books that other users can use after acknowledging reliance on the trust tag and thus on the super user. Via trust tags industrial users have hooked ACL2 to other tools in the design workflow, including SAT and SMT solvers.

Other verification environments have been built on top of ACL2. One notable one is the "ACL2 Sedan" theorem prover (ACL2s) which includes an Eclipse plug-in and provides a modern integrated development environment, supports several modes of interaction, provides a powerful termination analysis engine, includes a rich support for "types" and seamlessly integrates semi-automated bug-finding methods with interactive theorem proving [19].

I suspect that about half of our time on ACL2 has been devoted to supporting efficient execution of ACL2 functions and a convenient systems programming environment in which to develop trusted tools.

[19] ACL2's guard conjectures are akin to PVS' type correctness conditions ("TCCs") [48]. In both systems the prover is used as a type checker. A great deal of work went into the final design of ACL2's guards. The most basic question was "Are a function's guard part of its definitional equation or are the extra-logical?" We chose the latter to keep proofs simple. See [29].

8.4 Landmark Theorems

I listed in Section 1 some landmark theorems proved by ACL2. Some other theorems, dating from before 2000, are explained in the book *Computer-Aided Reasoning: ACL2 Case Studies* [30]. See also [56] and various links on the ACL2 home page [31], especially Publications, Workshops, and UT Seminar links.

But there are other kinds of important uses of ACL2 today.

One consequence of fast execution is that ACL2 models have dual use: they can be used as prototypes of an intended design and as formal specifications about which properties can be mechanically proved. For example, when AMD developed their RTL-to-ACL2 translator they ran 80 million floating point test cases through the ACL2 model of Athlon's `FMUL` and their own RTL simulator. However, the subsequent proof attempt exposed bugs not covered by the test suite. These bugs were fixed before the Athlon was fabricated. The same kind of dual use has been seen at Centaur and other hardware design groups.

Another consequence of fast execution and formal semantics is that useful verification tools can be built with ACL2, verified by ACL2, and then used independently of ACL2. The first example of this was in 1994, when Bishop Brock (with help from Warren Hunt, Kaufmann, and me) used ACL2 in 1993–94 to model and verify properties of a Motorola digital signal processing microcode engine. To prove that the engine implemented the microcode semantics Brock had to define a predicate recognizing hazard-free microcode sequences. Using that as a precondition he proved the engine correct. Thereafter, Motorola engineers ran the predicate to check microcode programs without necessarily realizing that the predicate had been mechanically verified to guarantee faithful execution of the microcode semantics.

Verification tools based on ACL2 (or parts thereof) have been built and have been used by groups at Intel and NXP, without users necessarily knowing ACL2 was involved. The Imandra prover, by Aesthetic Integration, Ltd., was closely modeled on ACL2 and is in use to verify financial algorithms [49] for compliance with relevant laws; ACL2 is used as an independent check on Imandra proofs.

Among the verified verification tools built with ACL2 include a metafunction for bounding intervals containing a machine address [46]; an efficient representation of symbolic states [45]; an integration of testing and theorem proving [20]; a tool for doing ACL2 termination analysis based on the Community Book repository [34]; a "bit-blaster" for proving finitely bounded ACL2 theorems with a verified BDD implementation or a SAT solver (whose claims can be checked by a verified ACL2 SAT proof-checker) [:DOC GL]. The ACL2 SAT proof-checker is now used routinely to check the claims of all participants in SAT competitions [25, 21]. Centaur runs its regressions at least nightly and has the ACL2 SAT proof-checker enabled once per week. ACL2(p) is an extension of ACL2 supporting parallelism in both proof and programming [:DOC parallelism].

8.5 Code

The ACL2 home page [31] explains how to obtain and install the source code and documentation. The code is distributed without cost under a 3-clause BSD license. The ACL2 Community Books (as well as a copy of source files) may be obtained from GitHub as explained on the ACL2 home page.

The source code runs on five different implementations of Common Lisp, one of which must be installed before ACL2 can be built on your machine. We have found that the variety of Lisps is useful for catching ACL2 bugs, and also serves to give users choices as various Lisps evolve or are superseded.

ACL2 has about 6 MB of source code plus about 5 MB of comments. In addition, there is about 6 MB of user-level documentation (not counting documentation for Community Books). For anybody studying how ACL2 works, design decisions, false starts, etc., the comments in ACL2 are well worth studying, especially the ones titled "Essay on . . ." covering about hundred topics. A developer's guide is available for experienced users wanting to better understand maintenance and development issues [:DOC developers-guide].

Approximately 90% of ACL2's source code is written in the ACL2 programming language. The remaining 10% is written in "raw" Common Lisp and mainly implements the features that we had to add to the applicative subset of Common Lisp to make it a convenient and efficient functional programming language.

9 Summary

If you walk at 3 miles/hour for 8 hours per day, 6 days a week, for two years – which is about the duration of an average research grant – you travel 14,976 miles or about 60% of the circumference of the Earth. If you walk for 48 years, you cover 359,424 miles, about one and a half times the distance to the Moon. The difference between PLTP and ACL2 is largely due to the fact that PLTP was a 2 year project and ACL2 has been a life's work.

10 Acknowledgments

I thank Bob Boyer, Matt Kaufmann, and Grant Passmore for their careful readings of early versions of this manuscript; all remaining mistakes are mine.

See [:DOC acknowledgements] for a list of important sponsors and contributors to the ACL2 project. I also thank the various readers of this history for their helpful reminders and corrections.

Bob Boyer and Matt Kaufmann are as much a part of this story as I am. I am incredibly lucky to have found two such research partners and I am deeply grateful to both of them. The users of Nqthm and ACL2, especially the students who joined us in the UT Tower in the 1980s deserve a great deal of thanks too. They pushed Nqthm to its limits. Many of those students then

moved to CLI with us and switched to ACL2 and have proceeded to push it, repeatedly, to its limits. ACL2 would not exist had it not been for these people. I am humbled and deeply grateful for their passion, patience, and persistence.

References

1. W. W. Bledsoe. Splitting and reduction heuristics in automatic theorem proving. *Artificial Intelligence*, 2:55–77, 1971.
2. W. W. Bledsoe and Peter Bruell. A man-machine theorem-proving system. *Artificial Intelligence*, 5:51–72, 1974.
3. R. 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(2):117–172, 1988.
4. R. S. Boyer, D. J. M. Davies, and J S. Moore. The 77-editor. Technical Report 62, Department of Computational Logic, University of Edinburgh, 1973.
5. R. S. Boyer and J S. Moore. The sharing of structure in theorem-proving programs. In *Machine Intelligence 7*, pages 101–116. Edinburgh University Press, 1972.
6. R. S. Boyer and J S. Moore. Proving theorems about pure lisp functions. *JACM*, 22(1):129–144, 1975.
7. R. S. Boyer and J S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.
8. R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, 1979.
9. 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*. Academic Press, London, 1981.
10. R. S. Boyer and J S. Moore. A verification condition generator for FORTRAN. In *The Correctness Problem in Computer Science*, pages 9–101, London, 1981. Academic Press.
11. R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.
12. R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In *Machine Intelligence 11*, pages 83–124. Oxford University Press, 1988.
13. R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*. Academic Press, New York, 1997.
14. Robert S. Boyer. *Locking: A Restriction of Resolution*. Department of Mathematics, University of Texas at Austin, 1971.
15. Robert S. Boyer and J S. Moore. On why it is impossible to prove that the BDX930 dispatcher implements a time-sharing system. In *Investigation, Development, and Evaluation of Performance Proving for Fault-Tolerant Computer Final Report, covering the period September 1978 to June 1982*, page Sections 14 and 15. Computer Science Laboratory, SRI International, Menlo Park, CA, July 1982.
16. Robert S. Boyer and Jr. Warren A. Hunt. Function memoization and unique object representation for ACL2 functions. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 81–89, New York, NY, USA, 2006. ACM.
17. R.S. Boyer, D.M. Goldschlag, M. Kaufmann, and J S. Moore. Functional instantiation in first-order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.
18. B. Brock and W. A. Hunt, Jr. Formal analysis of the motorola CAP DSP. In *Industrial-Strength Formal Methods*, pages 81–115. Springer-Verlag, 1999.
19. Harsh R. Chamarthi, Peter Dillinger, Pangiotis Manolios, and Daron Vroon. The acl2 sedan. Technical report, Northeastern University, Boston, 2019.

20. Harsh R. Chamarthis, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. Integrating testing and interactive theorem proving. In *10th International Workshop on the ACL2 Theorem Prover and its Applications*, volume 70, pages 4–19. Electronic Proceedings in Theoretical Computer Science, 2011.
21. Luís Cruz-Filipe, Marijn Heule, Warren Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified rat verification. In *26th International Conference on Automated Deduction (CADE 26)*, pages 220–236. Springer, 2017.
22. G. Gentzen. New version of the consistency proof for elementary number theory. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 132–213. North-Holland Publishing Company, Amsterdam, 1969.
23. Shilpi Goel, Jr. Warren A. Hunt, and Matt Kaufmann. *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*, pages 173–209. Springer, 2017.
24. David S. Hardin, Eric W. Smith, and William D. Young. A robust machine code proof framework for highly secure applications. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 11–20, New York, NY, USA, 2006. ACM.
25. Marijn Heule, Jr. Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In *Interactive Theorem Proving (ITP) 2017*, volume LNCS 10499, pages 269–284. Springer, 2017.
26. Michael Hiltzik. *Dealers of Lightning: Xerox PARC and the Dawn of the Computer Age*. Harper Collins, 1999.
27. L. Hodes. Solving problems by formula manipulation. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 553–559. British Computer Society, 1971.
28. Warren Hunt, Jr. Verifying VIA nano microprocessor components. In Roderick Bloem and Natasha Sharygina, editors, *FMCAD '10: Proceedings of the Formal Methods in Computer-Aided Design*, pages 3–10. ACM/IEEE, 2010.
29. M. Kaufman and J S. Moore. Design goals for ACL2. Technical Report CLI Tech Report 101, Computational Logic, Inc., August 1994. See `http://www.cs.utexas.edu/users/moore/publications/km94.pdf`.
30. M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.
31. M. Kaufmann and J S. Moore. The ACL2 home page. In *http://www.cs.utexas.edu/users/moore/acl2/*. Dept. of Computer Sciences, University of Texas at Austin, 2018.
32. Matt Kaufmann. An interactive enhancement to the boyer-moore theorem prover. In *Proc. 9th Intl. Conf. on Automated Deduction (CADE-9)*, volume LNCS 310, pages 735–736. Springer-Verlag, 1988.
33. Matt Kaufmann. An extension to the boyer-moore theorem prover to support first-order quantification. *Journal of Automated Reasoning*, 9(3):355–372, 1992.
34. Matt Kaufmann. Defunt: A tool for automating termination proofs by using the community books (extended abstract). In *15th International Workshop on the ACL2 Theorem Prover and its Applications*, volume 280, pages 161–163. Electronic Proceedings in Theoretical Computer Science, 2018.
35. Matt Kaufmann and J S. Moore. Limited second-order functionality in a first-order setting. *Journal of Automated Reasoning*, 2018.
36. J. C. King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, 1969.
37. Robert Kowalksi and Donald Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.
38. Hanbing Liu. *Formal Specification and Verification of a JVM and its Bytecode Verifier*. PhD thesis, University of Texas at Austin, 2006.
39. Donald MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust*. MIT Press, 2001.
40. Panagiotis Manolios and Daron Vroon. Algorithms for ordinal arithmetic. *Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science)*, 2741:243–257, 10 2003.
41. J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*. North-Holland Publishing Company, Amsterdam, The Netherlands, 1963.

42. J S. Moore. Computational logic: Structure sharing and proof of program properties. Ph.D. dissertation, University of Edinburgh, 1973. See `http://www.era.lib.ed.ac.uk/handle/1842/2245`.
43. J S. Moore. Automatic proof of the correctness of a binary addition algorithm. *ACM SIGARG Newsletter*, pages 13–14, 1975.
44. J S. Moore. Text editing primitives – the TXDT package. Technical Report CSL-81-2 (see `http://www.cs.utexas.edu/users/moore/publications/txdt-package.pdf`), Xerox PARC, 1981.
45. J S. Moore. Stateman: Using metafunctions to manage large terms representing machine states. In *ACL2 Workshop 2015*, volume 192, pages 93–109. EPTCS, 2015.
46. J S. Moore. Computing verified machine address bounds during symbolic exploration of code. In *Provably Correct Systems*, pages 151–172, 2017.
47. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages*, 1:245–257, 1979.
48. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752, Heidelberg, June 1992. Lecture Notes in Artificial Intelligence, Vol 607, Springer-Verlag.
49. Grant O. Passmore and Dennis Ignatovich. Formal verification of financial algorithms. In *Conference on Automated Deduction (CADE 26)*, volume 10395. Springer LNCS, 2017.
50. John Alan Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, January 1965.
51. David M. Russinoff. A case study in formal verification of register-transfer logic with acl2: The floating point adder of the amd athlon tm processor. In *Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume LNCS 1954. Springer, 2000.
52. David M. Russinoff. *Formal Verification of Floating-Point Hardware Design: A Mathematical Approach.* Springer, 2019.
53. R. Shostak. A practical decision procedure for arithmetic with function symbols. *JACM*, 26:351–360, 1979.
54. Anna Slobodova, Jared Davis, Sol Swords, and Jr. Warren Hunt. A flexible formal verification framework for industrial scale validation. In Satnam Singh, editor, *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 89–97. IEEE, 2011.
55. G. L. Steele, Jr. *Common Lisp The Language, Second Edition.* Digital Press, 30 North Avenue, Burlington, MA. 01803, 1990.
56. Jr. W. A. Hunt, M. Kaufmann, J S. Moore, and A. Slobodova. Industrial hardware and software verification with ACL2. In *Verified Trustworthy Software Systems*, volume 375. The Royal Society, 2017 (to appear). (Article Number 20150399).
57. Hao Wang. Toward mechanical mathematics. *IBM Journal of Research and Development*, 4(1):2–22, January 1960.
58. R. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence Journal*, 13(1):133–170, 1980.