ACL2 Induction Heuristics

J Strother Moore
Computer Science Department
University of Texas at Austin
Austin, Texas 78712 - 1757 USA
moore@cs.utexas.edu

December 23, 2020

1 Introduction

This article describes how ACL2 mechanizes mathematical induction.

The techniques, which involve three well-known mathematical ideas and a handful of heuristics, were first demonstrated in Boyer and Moore's Edinburgh Pure Lisp Theorem Prover (PLTP) of 1973, which was the first prover to attempt to automate induction in a general setting [10, 1]. Logically, the method depends on well-founded relations over some domain, a principle of definition admitting recursively defined functions justified by decreasing measures ensuring termination, and a principle of induction that is the precise dual of the principle of definition. The heuristics create plausible inductions for a conjecture and help choose among them. The generality of the mathematical formalizations and of the heuristics were then elaborated, first in Boyer and Moore's Thm and Nqthm provers [2, 3, 4] prover of the late 1970s and the 1980s and then in Kaufmann and Moore's ACL2 [7, 9] which was started by Boyer and Moore in 1989 but has been under continuous development by Kaufmann and Moore since 1992. ACL2 is in regular industrial use today [15].

This article narrowly focuses on ACL2 but is not intended specifically for the ACL2 audience. For a much deeper and broader discussion of the history of the mechanization of explicit and implicit[1] induction and various

[1]Implicit induction includes proof by consistency, implicit induction ordering where induction is over term structure rather than values, and lazy induction or *descente infinie* in which the system recogizes legal induction hypotheses when they arise rather than formulating them initially.

techniques in use by various mechanized provers see the article by Moore and Wirth [12].

The original Boyer–Moore techniques are described in detail and with examples on pages 163–208 of the 1979 book, *A Computational Logic* [2]. These early techniques are completely recognizable today in their modern incarnation in ACL2, but an historical account of the evolution from PLTP through Thm and Nqthm to ACL2 may be informative to some readers [11]. Two books about ACL2 [7, 6] describe how to use the system and some case studies of applications; these books contain many examples of inductions and also challenge the reader with exercises. We also refer to ACL2's extensive online user's manual which can be browsed under the The User's Manuals link on the ACL2 home page [9].

The reader interested in the nitty-gritty is urged to look at the ACL2 source code [9] which is available online without cost. The 6MB of documentation available there is aimed at users and does not cover how induction works, only how to override the automatic choices. The source code is written in the functional programming language that also serves as ACL2's logic, a side-effect-free subset Common Lisp [14]. The subset is formalized as a logic on pages 77–102 of *Computer-Aided Reasoning: An Approach* [7]. Henceforth in this article we refer to the previously mentioned book as simply as *An Approach*. Furthermore, the source code is heavily commented; for example of the 3,498 lines in the file `induct.lisp`, 1,356 are comments explaining the code. Our brief sketches below of the basic components of our induction machinery typically include a note naming a function in the ACL2 source code. All the functions mentioned are defined the file `induct.lisp`, online at `https://github.com/acl2/acl2/blob/master/induct.lisp`.

2 Preliminaries

We take for granted certain well-known concepts that must be formalized within the mathematical theory in use. ACL2's logical setting is intentionally at the weaker end of mathematical expressibility spectrum among mechanized computational logics: if it can be done in ACL2 it can almost certainly be done in other major systems. ACL2's logic is untyped first-order predicate calculus with induction and no explicit quantifiers. That basic setting pushes the user to define recursive functions to state theorems, which in turn underlies the system's inductive capabilities.

2.1 Well-Founded Relations

Let $\prec$ be a *well-founded* relation over some domain $\mathbb{D}$. By well-founded we mean there is no infinitely descending sequence of elements $x_1, x_2, x_3, \cdots \in \mathbb{D}$ such that $\cdots \prec x_3 \prec x_2 \prec x_1$. For example, the familiar less-than operator, $<$, is well-founded over $\mathbb{N}$, the set of natural numbers.[2]

2.2 Principle of Definition

The purpose of a principle of definition is to allow the user to extend the logical theory by the addition of an axiom defining a new function symbol *while preserving the consistency of the theory.* A new axiom of the form $f(x) = b$ is suggestive of a definition of f, at least if f is a *new* function symbol, i.e., one mentioned in no other axiom. But if the axiom $f(x) = 1 + f(x)$ is added to a theory that includes the usual axiomatization of arithmetic then one can prove anything: it is a theorem that no number is equal to its own successor. The least one can expect of a definitional principle is that the only "new" theorems one can prove after defining f are formulas that somehow (perhaps ancestrally through a function used in the definition) involve f. Any formula not ancestrally dependent on f that is provable after defining f is provable in the theory before f was introduced. Logicians call such extensions *conservative.*

Roughly speaking, the ACL2 definitional principle deals with proposed definitions $f(v_1, \ldots, v_n) = b$. We call the v_i the *formals* of (the definition of f) and b the *body.* We call $f(a_1, \ldots, a_n)$ a *call* of f and the a_i are the *actuals.* If γ is a call of f, then we let σ_γ be the variable substitution replacing the formals by their respective actuals. Finally, if δ is a term or formula and σ is a substitution replacing v_i by a_i, then δ/σ is the term or formula obtained by applying σ to δ, i.e., uniformly replacing free occurrences of the v_i in δ by the corresponding a_i.

Without loss of generality we insist that b is a nest of if-then-else expressions, each expression having a *test*, a *true-branch*, and a *false-branch.* We can explore this as a tree and collect the tests (positive or negative) we pass as we take various branches. We stop the descent down a branch (and continue exploring other branches) when we encounter either an if-then-else containing a call of f somewhere in its test or when we encounter a term

[2] PLTP used $<$ and $\mathbb{N}$ for $\prec$ and $\mathbb{D}$. Thm used lexicographic orderings over n-tuples of naturals. Nqthm and ACL2 use the less-than relation over the ordinals below $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$, where ω is the smallest infinite ordinal. See pp. 85–88 of *An Approach* however the representation of the ordinals has changed since the book was published. See the topic "ordinals" in the online ACL2 user's manual.

other than an if-then-else. We call those stopping points the *tips* of b.[3] Let $\mathbb{T}$ be the set of all tips of the proposed definitional equation. Every tip, $t \in \mathbb{T}$, is characterized by the conjunction of the positive and negative tests ruling it, Π_t, and the set $\mathbb{S}_t$ of variable substitutions $\{\sigma_{\gamma_1}, \ldots, \sigma_{\gamma_m}\}$ for the recursive calls, $\{\gamma_1, \ldots, \gamma_m\}$ of f occurring in the tip.[4] The set of pairs $\{\langle \Pi_t, \mathbb{S}_t \rangle | t \in \mathbb{T}\}$ containing the tests leading to and substitutions derived from each tip is called the *machine* for f.

The definitional principle imposes both syntactic requirements and proof obligations on a proposed definitional equation. The syntactic requirements are: f is a new function symbol, the formals are distinct variable symbols, and the body is a term whose free variables are among the formals.

The proof obligation is as follows. There must exist a term, μ, called the *measure*, such that the following is a theorem

$$(\mu \in \mathbb{D}) \wedge \left(\bigwedge_{t \in \mathbb{T}} \left(\Pi_t \rightarrow \left(\bigwedge_{\sigma \in \mathbb{S}_t} (\mu/\sigma \prec \mu) \right) \right) \right)$$

That is, for a definition to be *admitted* as a new axiom we must first be able to prove that there is a measure of the arguments that decreases in a well-founded sense in every recursive call.[5] This guarantees termination and along with the syntactic restrictions guarantees the existence of a function satisfying the definitional equation.

Only after checking the syntactic restrictions and using the prover to prove the above measure theorem does ACL2 add the new axiom

$$f(v_1, \ldots, v_n) = b.$$

ACL2 also stores the machine for future use by the induction mechanism.

This definitional principle is conservative.[6] See pp. 89–92 of *An Approach* and for more details "Structured Theory Development for a Mechanized Logic," by Kaufmann and Moore [8].

Finally, note that if a subsequent conjecture mentions a call, γ, of f then equality is maintained by replacing γ by b/σ_γ. We call this *expanding* the call. Because f may involve recursive calls, heuristics must control expansion.

[3] The ACL2 exploration of b is a little more sophisticated in how it recognizes tips. See the discussion of "rulers" in the user's manual.

[4] We sometimes treat Π_t as a term and sometimes as the set of its conjuncts.

[5] ACL2 requires the user to supply μ with the proposed definition if a simple heuristic fails to guess an appropriate measure.

[6] This is not to say that ACL2's prover could discover the alternative proofs guaranteed by conservativity.

2.3 Principle of Induction

An induction principle can be formulated that is precisely the dual of the principle of definition. It is just a weaker version of the principle of well-founded induction, weaker because we only assume a finite number of smaller instances of the conjecture instead of all smaller instances. See pp. 93–98 of *An Approach*. Rather than state the principle here we just explain how a recursive definition as admitted above suggests an induction. Suppose f, with formals $v_1, \ldots, v_n$ was admitted with measure μ and body b.

The theorem proved at admission establishes that μ always returns an element of $\mathbb{D}$. Furthermore, for each tip $t \in \mathbb{T}$, the measure decreases in a well-founded sense in every recursive call in t, provided the ruling conjunction Π_t is true.

This leads to the following possible induction scheme to prove φ, which, for the sake of simplicity here, we assume includes the formals of f among its free variables.

$$\bigwedge_{t \in \mathbb{T}} \left(\left(\Pi_t \wedge \bigwedge_{\sigma \in \mathbb{S}_t} \varphi/\sigma \right) \rightarrow \varphi \right).$$

Note that for any tip t with no recursive calls, the conjunct for t reduces to $\Pi_t \rightarrow \varphi$, i.e., we must prove φ assuming only the tests leading to that tip. These are all *base cases* of the induction scheme.

For any tip t containing recursive calls, the conjunct for t requires us to prove that the tests leading to that tip imply φ but we also get to assume φ/σ_γ for every recursive call γ in the tip. These are the *induction steps*, the φ/σ_γ are the *induction hypotheses* and φ is the *induction conclusion*.

Consider a simple recursive definition that recurs down to 0 by subtracting 1 from a natural number formal, n. Let the measure μ be the identity function on n and the well-founded relation be less-than on the naturals.[7] To prove $\varphi(n)$ (for all $n \in \mathbb{N}$) the suggested induction principle is

$$(n = 0 \rightarrow \varphi(n)) \wedge (n \neq 0 \wedge \varphi(n-1) \rightarrow \varphi(n))$$

as opposed to the more common but equivalent

$$\varphi(0) \wedge (\varphi(n) \rightarrow \varphi(n+1)).$$

[7] Technically, since the measure must always produce a result in the domain of the well-founded relation, the appropriate measure here is `nfix` which is the identity on naturals and 0 everywhere else. In a suitably typed system that would not be necessary. We ignore such issues in this article.

2.4 General Proof Strategy

A common proof strategy, and the one generally followed by ACL2, is to first try to simplify the goal, φ. Simplification focuses on normalizing terms, both by using rewrite rules derived from previously proved theorems and by applying various decision procedures.

But if simplification fails to prove φ, perhaps induction is warranted. In that case, the first step is to select an induction scheme, apply the scheme to φ to get a new set of goals, and recursively try to prove each via simplification and the other techniques.

But weak conjectures are often impossible to prove by induction because no available induction hypothesis can be brought to bear on the conclusion. A classic example is provided by a function $f(n, m)$ that recurs by decrementing n and incrementing m, returning m when $n = 0$. One then might wish to prove $f(n, 0) = n$. But that will not yield to induction. Instead, one should prove $f(n, m) = n + m$ and then prove the "main" theorem by instantiation and algebraic simplification. PLTP and the other provers in the family all attempt to generalize before settling on induction, but generalization often requires more creativity than our provers have. To deal with this problem Thm, Nqthm, and ACL2 allow the user to state lemmas, which are often proved by induction, and which then may be used in the proofs of other theorems.

The control flow of ACL2 is: simplify (which including decision procedures and previously verified proof tools), eliminate "destructor" function symbols by introducing "constructors", use and eliminate equalities (a generalization step) stemming from previous inductions, generalize certain common subexpressions, and eliminate irrelevant hypotheses. All of these techniques are informed by previously proved lemmas. If these techniques do not prove the conjecture, then ACL2 tries induction, generates the new subgoals, and recurs to prove them all. This scheme is called the *waterfall*. See pp. 121–124 of *An Approach* for more details.[8]

Observe that if φ contains a call of f, $f(v_1, \ldots, v_n)$ (for simplicity, again, where the actuals of the call are the formals of f) and we choose to induct according to the scheme suggested by f, then when the simplifier expands that call of f in the conclusion of a base case the result is a term not involving

[8] Although it works in a simpler logical setting, the prover described in the 1979 book Boyer–Moore book *A Computational Logic* [2] used variants of these same techniques and Chapters V–XIII of that book describe them in detail with a running example. To see the original incarnation of the techniques, which did not include elimination of either destructors or of irrelevance, see Moore's 1973 dissertation [10] or the much shorter Boyer–Moore article of 1975 [1].

f. When the call is expanded in an induction step we will find an induction hypothesis about each recursive call introduced.

So one should try to select an induction suggested by terms in φ that when expanded produce smaller instances of themselves without instantiating variables in terms that are not expanded. The last qualification is motivated by the following observation: Since each induction hypothesis differs from the induction conclusion only by the application of a substitution, any term in the conclusion that does not involve a variable changed by the substitution will reoccur identically in the hypothesis.

We do not guarantee this induction strategy will succeed, but it is a powerful heuristic observation.

In summary: Do not try induction until other methods have failed. And when you do try induction, try to select a scheme as above.

3 Collecting Suggested Inductions

If induction is to be tried, we first explore φ and look for calls of recursive functions. So consider a term γ calling f. The restriction mentioned above (that the actuals of the call of f be the formals of f) can be relaxed.

We say a_i in a call of f, $(fa_1 \dots a_n)$, is in a *changeable position* if the corresponding formal, v_i is used in the measure admitting f and some substitution of f's machine replaces v_i by something different from v_i. We say a_i is in an *unchangeable position* if v_i is measured and every substitution in the machine replaces v_i by v_i.

A call γ of f *suggests* an induction if (i) every actual of γ in a changeable position is a distict variable, and (ii) no free variable of any actual in an unchangeable position occurs among the variables identified in (i). The set of variables in changeable positions of γ are called the *changeables* of γ and the set of variables occurring in terms in unchangeable positions of γ are called the *unchangeables* of γ.

If f suggests an induction the machine for f is instantiated so that it uses the variables of γ (instead of the formals of f). Any substitution pairs changing unchangeables or attempting to substitute for non-variables are deleted. The result is a new machine derived from γ. For example, suppose $f(v_1, v_2, v_3)$ in recursion counts the natural v_1 up by 1 until it exceeds the natural v_2 and changes v_3 to $v_1 * v_3$ (so that the natural difference between v_2 and v_1 decreases in recursion). The first and second formals are measured; the third is not. Now consider the term $\gamma = f(x, y+z, z)$ which might occur in φ. It suggests an induction in which x is replaced by $x+1$, and y and z

are held fixed. The substitution pair that would replace z by $x * z$ is deleted because it violates the restriction that the second argument be unchanged by the substitution. The measure theorems proved during the admission of f suffice to prove that this induction is sound, so no induction-time work needs to be done to approve this scheme.

The induction *candidate* suggested by γ is the machine just described but augmented with some metadata including the list of *controllers* (all variables in γ involved in the measure, e.g., x, y, and z in the example above), the changeable and unchangeable variables, the original justification of the function suggesting this induction, the set of terms suggesting this induction (initially just $\{\gamma\}$ and henceforth called the *suggesters*), and a score meant to indicate how closely the machine matches the recursion in f (initially the number of measured formals of f divided by the arity of f).

For the nitty-gritty see the function `induct`. It calls the other functions mentioned below. For enforcing the restrictions on which actuals are variables see `sound-induction-principle-mask`. For gathering the candidates, see `get-induction-cands-from-cl-set`.

Once all the candidates have been gathered, we apply a series of filters and transformers to create (what is hoped to be) an appropriate induction scheme.

4 Flushing

We say candidate c_1 is *subsumed* by candidate c_2 if the changeable variables of c_1 are a subset of those of c_2, the unchangeable variables of c_1 are a subset of those of c_2, and every entry of c_1's machine can "fit" in an entry in c_2's machine. Roughly speaking this means that for every $\langle \Pi_{t_1}, \mathbb{S}_{t_1} \rangle$ in c_1 there is a pair $\langle \Pi_{t_2}, \mathbb{S}_{t_2} \rangle$ in c_2 such that $\Pi_{t_1} \subseteq \Pi_{t_2}$ and for every substitution $\sigma_1 \in \mathbb{S}_{t_1}$ there is a $\sigma_2 \in \mathbb{S}_{t_2}$ such that $\sigma_1 \subseteq \sigma_2$. However, we require that each substitution from c_2 be used to absorb at most one substitution from c_1.

If c_1 is so subsumed, we *flush* it down c_2, discarding c_1 after changing the metadata in c_2 by adding c_1's score to that of c_2, unioning the suggesters of c_1 into those for c_2 and analogously unioning the other fields. For the nitty-gritty see `flush-candidates`.

5 Merging

We say c_1 is *mergeable* with c_2 if there is a non-empty intersection between their changeable variables, there are empty intersections between the changeable variables of c_1 and the unchangeable variables of c_2 and vice versa, and it is possible to merge the machine of c_1 into that of c_2. Intuitively, we try to extend each substitution of c_2 with a substitution of c_1 that agrees with it on changed variables. We do not change the tests ruling the substitutions in c_2. We pay special attention to the agreement of changes to c_2's measured variables so the justification of the extended c_2 is the same as that of the original. Merging adds scores and unions the other fields into c_2.

The idea is that if, under some tests, c_2 replaces x by δ_x and y by δ_y, and under syntactically compatible tests c_1 replaces x by δ_x and z by δ_z, then the two candidates agree on x and do not conflict on y and z. In the extended c_2, x, y, and z are replaced, respectively, by δ_x, δ_y and δ_z. For example, if $f(v_1, v_2)$ is defined to decrement each argument simultaneously in recursion and φ contains both $f(x, y)$ and $f(y, z)$ (giving rise to two mergeable candidates), then the merged candidate inducts on x, y, and z simultaneously. If we induct only as suggested by, say, $f(x, y)$ then the instance of $f(y, z)$ in the induction hypothesis will not match the $f(y, z)$ in the induction conclusion whether we expand $f(y, z)$ or not. For the nitty-gritty see `merge-tests-and-alists-lsts`.

6 Vetoing

The purpose of the next step is to eliminate candidates that are flawed. The basic idea is that a candidate is *flawed* if there is competition from other candidates about how to treat one or more of its variables. For example, if $f(x, y)$ recursively decrements x while not changing y, then if both $f(x, y)$ and $f(y, z)$ occur in φ then we try to avoid the induction suggested by $f(y, z)$ because its induction hypothesis would then contain $f(x, y-1)$ which would not match the $f(x, y)$ in the induction conclusion. On the other hand, the induction suggested by $f(x, y)$, which does not change y, leaves $f(y, z)$ unchanged in both the induction hypothesis and the induction conclusion.

We use two heuristics. First we throw out any candidate whose changeable variables have a non-empty intersection with either the changeable or unchangeable variables of another. If that throws out all candidates we revert the set of candidates and throw out only those whose changeable variables have a non-empty intersection with the unchangeables of another

candidate. If that also throws out all candidates, we revert again and skip this heuristic. For the nitty-gritty see `compute-vetoes`.

7 Voting

Should there still be multiple candidates, we vote in two rounds. First we select the candidates with the maximal "complexity," where *complexity* is just the number of suggesters that fail to satisfy a restricted syntactic recognizer for primitive-recursion. (Recall that flushing and merging both accumulate the suggesters, so that by the time voting occurs each surviving candidate may have several suggesters.) Successful inductions, followed by simplification, etc., tends to eliminate suggesters from the subsequent subgoals. We prefer to eliminate the most complicated functions as soon as that appears possible; otherwise the opportunity may lost.

Should there be multiple candidates after choosing the most complex ones, we select those with the maximal score. The final tiebreaker is to just choose the first candidate in the list of survivors. For the nitty-gritty see `maximal-elements`.

8 Communicating to the Simplifier

After selecting an induction and applying it to the goal in question to get a set of base cases and induction steps, those goals are sent back over the waterfall. But induction communicates the suggesters to the simplifier. By default, the simplifier preferentially expands those terms.[9] In addition, the basic heuristics for controlling the expansion of recursive functions is sensitive to the question "do the terms introduced by a (tentative) expansion already occur in the goal?" If so, the expansion is allowed. Since induction hypotheses will include the recursive calls introduced by the suggesters, and since the simplifier attempts to normalize terms, this basic heuristic tends eventually to expand the suggesters even if their subterms have been rewritten (which could prevent the simplifier from recognizing them in the communication from the induction mechanism).

[9] As with many ACL2 heuristics, the user can override these preferences with hints.

9 Caveats

These heuristics do not always choose an induction that proves the theorem, even when such an induction exists. For that reason one must provide a means for specifying the appropriate induction, which is done with a user-provided hint in ACL2 that means "induct as suggested by the term γ."

One source of inappropriate inductions in ACL2 stems from our failure to implement *rippling* [5]. For example, suppose $f(x)$ measures x and recurs to $1 + f(d(x))$ and $g(y, z)$ measures y and recurs to $g(y - 1, a(z))$. Then ACL2 only sees one induction suggested by terms in $g(f(x), z)$, namely the one suggested by $f(x)$, in which x is replaced by $d(x)$ in the induction hypothesis. The g term does not suggest an induction because its first (changing and measured) argument is not a variable. But rippling-out and -in could suggest replacing x by $d(x)$ and z by $a(z)$ by noting the effect of expanding f on the enclosing g term and the consequent changing of z. The ACL2 user has to explicitly define a function to recur that way and then provide it as a hint.

10 Performance

In this section we indicate how successful these heuristics are, by analyzing the proofs in a large suite of theorems.

The test suite contains 467,892 named theorems. The suite consists of almost all of the ∼7,500 books in ACL2's Community Books repository, contributed by many different users. About a dozen books were omitted. The omitted books contain checks to confirm that the prover output was as expected and the statistics gathering process changed what is displayed. However, the vast majority of these named theorems are mechanically generated with macros implemented by power users who provide tools to do common things like introduce new data structures or prove common classes of theorems in a given domain by well-understood proof tactics. Furthermore, many books of theorems are available that make reasoning in certain domains automatic via lemma-driven simplification.

The end result is that induction is used in the regression only 72,626 times.

When ACL2's induction heuristics fail, users provide explicit induction hints. A total of 27,322 induction hints were used, meaning the heuristics described here chose appropriately at least 62% of the time.[10]

[10] Of course, the conjectures in question are not "random." Users strive to state theorems

So now we characterize the contributions of the various heuristics.

In the 45,304 cases where an induction was selected without hints from the user, a total of 104,197 candidates were suggested. So about 2.29 candidates were suggested per conjecture.

Flushing eliminated about 16% of those candidates, and merging eliminated about 33% of the remaining ones, leaving 57,700 candidates.

But 15% of those surviving candidates were vetoed by others, leaving 49,961 candidates to be subjected to voting.

Note that if we use voting among 49,961 candidates to choose 45,304 induction winners, we see that at least 90% of the time there is only one candidate.

Choosing the most complex candidates winnows the field to 48,648 and choosing the highest scoring among those cuts it to 45,539. So the final tiebreaker is used at most 235 times out of 45,304 inductions.

Induction is sometimes used multiple times within a single proof. While 64,357 proofs used just one induction, 2,480 proofs used more than one. For example, 1,407 proofs used two inductions and 634 proofs used three. Sometimes ACL2 will prove a theorem with multiple inductions, perhaps without the user even noticing. For example, 12 proofs used ten inductions, two used 19, and one proof used 30. The most inductions done in any proof is 516.

The theorem using 516 inductions arises in the admission of a function defining the operational semantics of Clocked Control/Data FLow Graphs and is part of a bigger project to create a certified loop pipelining algorithm [13]. Initial simplification of the measure theorem splits into many subgoals, most of which are proved by simplification, but 12 of those subgoals cannot be proved by ACL2's simplifier so 12 more inductions are done. Some of those inductions generate additional subgoals to prove inductively. The maximum number of inductions along any one branch of that proof tree is three. The authors probably could have avoided many of these inductions by proving appropriate lemmas or otherwise giving hints to the prover. However, the proof takes only 18.40 seconds on a modern (2019) laptop. So why bother?

---

the prover can handle by itself. On the other hand, many user-defined macros formalizing common proof schemes insert induction hints that might suggest the same inductions ACL2 would choose on its own.

11 Summary

Induction cannot be isolated from the rest of the prover. The definitional principle and the use of recursive functions to express properties are key to the successful "automation of induction." But so is the simplifier and other techniques not discussed here. Indeed, PLTP was entirely designed around induction and tried to discover all necessary lemmas. The subsequent provers acquired other features largely to support more general use. In particular, those subsequent provers supported the use of previously proved lemmas so the user could build domain-specific libraries capable of dispatching most goals with rewriting, linear arithmetic, etc. But the lemmas in those libraries can often only be proved by induction.

12 Acknowledgments

Robert S. Boyer and I met in 1971 in Edinburgh and began this project. Roughly a quarter century later, Matt Kaufmann joined the team and eventually became, with me, the co-developer of ACL2. So this article reports joint work by three people over half a century. There are several lessons: First, the brevity of this chapter belies the true complexity of the underlying code; that is why I encourage inspection of ACL2's source code. And second, my debt to Bob and Matt cannot be overestimated. I also owe thanks to too many sponsors to list and to the user communities of our provers who constantly push the envelope and raise important problems to think about.

References

[1] R. S. Boyer and J S. Moore. Proving theorems about pure lisp functions. *JACM*, 22(1):129–144, 1975.

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

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

[4] R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition.* Academic Press, New York, 1997.

[5] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-Level Guidance for Mathematical Reasoning.* Cambridge

Tracts in Theoretical Computer Science, Cambridge University, UK, 2005.

[6] M. Kaufmann, P. Manolios, and J S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, Boston, MA., 2000.

[7] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.

[8] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[9] 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, 2020.

[10] 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`.

[11] J Strother Moore. Milestones from the pure lisp theorem prover to acl2. *Formal Aspects of Computing*, 2019.

[12] J Strother Moore and Claus-Peter Wirth. Automation of mathematical induction as part of the history of logic. *IfCoLog Journal of Logics and their Applications*, 4(5):1505–1634, 2017.

[13] Disha Puri, Sandip Ray, Kecheng Hao, and Fei Xie. Mechanical certification of loop pipelining transformations: A preview. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 549–554, Cham, 2014. Springer International Publishing.

[14] G. L. Steele, Jr. *Common Lisp The Language, Second Edition*. Digital Press, 30 North Avenue, Burlington, MA. 01803, 1990.

[15] 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. (Article Number 20150399).