ACL2: An Industrial Strength Version of Nqthm

Matt Kaufmann*and J Strother Moore
Computational Logic, Inc.†
1717 West Sixth Street, Suite 290
Austin, TX 78703-4776, USA

June, 1996

Abstract

ACL2 is a reimplemented extended version of Boyer and Moore's Nqthm and Kaufmann's Pc-Nqthm, intended for large scale verification projects. However, the logic supported by ACL2 is compatible with the applicative subset of Common Lisp. The decision to use an "industrial strength" programming language as the foundation of the mathematical logic is crucial to our advocacy of ACL2 in the application of formal methods to large systems. However, one of the key reasons Nqthm has been so successful, we believe, is its insistence that functions be total. Common Lisp functions are not total and this is one of the reasons Common Lisp is so efficient. This paper explains how we scaled up Nqthm's logic to Common Lisp, preserving the use of total functions within the logic but achieving Common Lisp execution speeds.

1 History

ACL2 is a direct descendent of the Boyer-Moore system, Nqthm [8, 12], and its interactive enhancement, Pc-Nqthm [21, 22, 23]. See [7, 25] for introductions to the two ancestral systems. "ACL2" stands for "A Computational Logic for Applicative Common Lisp."

Like Nqthm, ACL2 supports a Lisp-like, first order, quantifier free mathematical logic based on recursively defined total functions. Experience with the

*Matt Kaufmann's address is now Motorola Inc., Bridgepoint Plaza I, 5918 W. Courtyard Dr., Suite 330, Austin, TX 78730.

†The theorem prover used in this work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Order 7406, and the Office of Naval Research, Contract N00014-94-C-0193. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency, the Office of Naval Research, or the U.S. Government.

earlier systems [3, 4, 5, 8, 10, 11, 13, 14, 15, 18, 19, 20, 29, 31, 32, 35, 36, 37, 38, 41, 42, 43] supports the claim that such a logic is sufficiently expressive to permit one to address deep mathematical problems and realistic verification projects. The fact that the Nqthm logic is executable is also an important asset when using it to model hardware and software systems: the models can be executed as a means of corroborating their accuracy. Consider for example [2] where an Nqthm model of the MC68020 is corroborated against a fabricated chip by running 30,000 test vectors through the Nqthm model.

Some of the largest formal verification projects done so far have been carried out with Nqthm. We cite explicitly the CLI short stack [4], the design and fabrication of the FM9001 microprocessor [20], and the verification of the Berkeley C string library on top of the MC68020 microprocessor [13, 43]. The formal models in these projects are collectively several hundred pages long and involve many functions. Despite such successes, Nqthm was not designed for these kinds of large-scale projects and it has several inadequacies. The most important inadequacy of Nqthm is its lack of theorem proving power: if it would quickly settle every question put to it, one could proceed more efficiently. While we are always looking for better proof techniques (e.g., [33]), we do not know how to build a *significantly* more powerful and automatic theorem prover for Nqthm's logic.[1] Therefore, to "scale up" Nqthm we focused on engineering issues.

We decided that a good first step would be to adopt as a logic the applicative subset of a commonly used programming language, thereby gaining access to many efficient execution platforms for models written in the logic and many program development (i.e., modeling) environments. We chose Common Lisp because of its expressiveness, efficiency and familiarity. Properly declared Common Lisp can execute at speeds comparable to C.

Three guiding tenets of the ACL2 project have been to (1) conform to all compliant Common Lisp implementations, (2) add nothing to the logic that violates the understanding that the user's input can be submitted directly to a Common Lisp compiler and then executed (in an environment where suitable ACL2-specific macros and functions — the *ACL2 kernel* — are defined), and (3) use ACL2 as the implementation language for the ACL2 system.

The third tenet is akin to recoding Nqthm in the Nqthm logic, a task that we believe would produce unacceptably slow performance. Programming the ACL2 system in ACL2 repeatedly forced us to extend the subset so that we could write acceptably efficient code. Several iterations of the system were built. The current system consists of over 5 megabytes of applicative source

[1] We emphasize the word "significantly" here because ACL2's theorem prover is in fact more powerful than Nqthm in many ways. For example, ACL2 supports faster propositional calculus, an OBDD facility, forward chaining similar to that in [16], congruence-based rewriting, and "forcing" — the last being a very useful technique in which the system assumes a hypothesis of a rule for the purposes of applying the rule and later devotes all of its resources to proving that the hypothesis true. Nevertheless, these improvements do not give ACL2 the ability to settle quickly every question put to it!

code, including documentation.

The first version of the system was written in the summer and fall of 1989, by Boyer and Moore. As time went by, Boyer's involvement decreased and Kaufmann's increased. Eventually Boyer decided he should no longer be considered a coauthor. ACL2 has been used in modeling and verification projects within Computational Logic, Inc. (CLI), for several years. We released the first public version of ACL2 in September, 1995.

To obtain ACL2 by ftp, first connect to ftp.cli.com by anonymous login. Then 'cd' to /pub/acl2/v1-8, 'get README' and follow the directions therein. Alternatively, one may obtain ACL2 via the World Wide Web with the URL http://www.cli.com/software/acl2/index.html. The ACL2 system includes extensive hyper-linked on-line documentation available via ACL2 documentation commands, HTML browsers, Emacs Info, and in hardcopy form. See [27].

Acknowledgements. Without Bob Boyer's contributions, especially in the initial stages of ACL2 development, it is quite possible that ACL2 would never have been created. We are also especially grateful to Bishop Brock for providing valuable feedback based on his heavy use of ACL2 for many months. Bill Young improved and contributed tutorial material to the documentation. Art Flatau, Noah Friedman, and Laura Lawless contributed to technical aspects of the documentation and provided useful feedback. We also acknowledge contributions from Bill Bevier, Alessandro Cimatti, Rich Cohen, John Cowles, Warren Hunt, Jun Sawada, Bill Schelter, Mike Smith. Finally, we are grateful for the utility provided by Gnu Emacs; in particular, Richard Stallman provided a Texinfo patch to assist in building the documentation.

2 The ACL2 Logic

The definition of Common Lisp used in our work has been [39, 40]. We have also closely studied [34].

The ACL2 logic is a first-order, quantifier-free logic of total recursive functions providing mathematical induction on the ordinals up to ϵ_0 and two extension principles: one for recursive definition and one for constrained introduction of new function symbols. We sketch the logic here.

2.1 Syntax

The syntax of ACL2 is that of Common Lisp. Formally, an ACL2 term is either a variable symbol, a quoted constant, or the application of an n-ary function symbol or lambda expression, f, to n terms, written (f $t_1 ... t_n$). We show the syntax for the primitive constants below. This formal syntax is extended by a facility for defining constant symbols and macros.

2.2 Rules of Inference

The rules of inference are those of Nqthm, namely propositional calculus with equality together with instantiation and mathematical induction up to ϵ_0. Two extension principles, recursive definition and encapsulation, are also provided; these are discussed in subsection 2.6.

2.3 Axioms for Primitive Data Types

The following primitive data types are axiomatized.

- **ACL2 Numbers**. The numbers consist of the rationals and complex numbers with rational components. Examples of numeric constants are `-5`, `22/7`, and `#c(3 5)`.
- **Characters**. ACL2 supports 256 distinct characters, including Common Lisp's "standard characters" such as the character constants `#\A`, `#\a`, `#\,` `#\Newline`, and `#\Space`.
- **Strings**. ACL2 supports strings of characters, e.g., the string constant `"Nonnumeric argument"`.
- **Symbols**. ACL2 supports many of Common Lisp's symbols. Symbols are objects consisting of two strings: a package and a name. The symbol constant `EXEC` in the package `"MC68020"` is written `MC68020::EXEC`. By convention, one package is always selected as "current" and its name need not be written. Thus, if `"MC68020"` is the current package, the symbol above may be more simply written as `EXEC`. Packages may "import" symbols from other packages (although in ACL2 all importation must be done at the time a package is defined). If `MC68020::EXEC` is imported into the `"STRING-LIB"` package then `STRING-LIB::EXEC` is in fact the same as `MC68020::EXEC`.
- **Lists**. ACL2 supports arbitrary ordered pairs of ACL2 objects, e.g., the list constant `(X MC68020::X ("Hello." (1 .  22/7)))`.

2.4 Axioms Defining Other Primitive Function Symbols

Essentially all of the Common Lisp functions on the above data types are axiomatized or defined as functions or macros in ACL2. By "Common Lisp functions" here we mean the programs specified in [39] or [40] that are (i) applicative, (ii) not dependent on state, implicit parameters, or data types other than those in ACL2, and (iii) completely specified, unambiguously, in a host-independent manner. Approximately 170 such functions are axiomatized.

Common Lisp functions are partial; they are not defined for all possible inputs. But ACL2 functions are total. Roughly speaking, the logical function of a given name in ACL2 is a completion of the Common Lisp function of the same name obtained by adding some arbitrary but "natural" values on arguments outside the "intended domain" of the Common Lisp function. ACL2 requires that every ACL2 function symbol have a *guard*, which may be thought of as a predicate on the formals of the function describing the intended domain. The guard on the primitive function `car`, for example, is `(or (consp x) (equal x nil))`, which requires the argument to be either an ordered pair or `nil`. But guards are entirely extra-logical: they are not involved in the axioms defining functions. We discuss the role of guards and the relation between ACL2 and Common Lisp later.

2.5 Axioms for Additions to Common Lisp

To applicative Common Lisp we add four important new features by introducing some new function symbols and appropriate axioms.

- We add new multiply-valued function call and return primitives that are syntactically more restrictive but similar to the Common Lisp primitives `multiple-value-bind` and `values`. Our primitives require a function always to return the same number of values and to be called from contexts "expecting" the appropriate number of values. These restrictions allow our multiply-valued functions to be implemented more efficiently than Common Lisp's (at least in the case of Gnu Common Lisp). Logically speaking, a vector of multiple values returned by a function is just a list of the values; but the implementation is more efficient because the list is not actually constructed.

- We add an explicit notion of "state" to allow the ACL2 programmer to accept input and cause output. The input/output functions of Common Lisp are not in ACL2 because they are not applicative: they are dependent on an implicit notion of the current state. An ACL2 state is an n-tuple containing, among other things, the file system and open input/output "channels" to files. Primitive input/output functions are axiomatized to take a state as an explicit parameter and to return a new state as an explicit result (usually one of several results). Syntactic checks in the language insure that the state is single-threaded, e.g., if a function takes state as an argument and calls a function that returns a new state, the new state (or more precisely, the final descendent of it) must be returned. This gives rise to a well-defined notion of the "current state" which is supplied to top-level calls of state dependent ACL2 functions. The state returned by such calls becomes, by definition, the next current state. Because of these restrictions, the execution of a state dependent function need not (and does not) actually construct new state n-tuples but literally modifies the underlying Common Lisp state.

- We add fast applicative arrays. These are implemented, behind the scenes, with Common Lisp arrays in a manner that always returns values in accordance with our axioms and operates efficiently provided certain programming disciplines are followed (namely, they are used in a single-threaded way so that only the most recently updated version of an array is used). There is no syntactic enforcement of the discipline; failure to follow it simply leads to inefficient execution and warning messages.

- We add fast applicative property lists in a manner similar to that for arrays.

2.6 Extension Principles

Finally, ACL2 has two extension principles: definition and encapsulation. Both preserve the consistency of the extended logic. See [26]. Indeed, the standard model of numbers and lists can always be extended to include the newly introduced function symbols. (Inconsistency can thus be caused only if the user adds a new axiom directly rather than via an extension principle.) The definitional principle insures consistency by requiring a proof that each defined function terminates. This is done, as in Nqthm, by the identification of some ordinal measure of the formals that decreases in recursion. In [8] we show (for Nqthm) that this insures that one and only one set-theoretic function satisfies the recursive definition, and that proof carries over to the ACL2 case, with appropriate treatment of the nonuniqueness of the constrained functions used in the definition. The encapsulation principle preserves consistency by requiring the exhibition of witness functions that have the alleged properties. We do not further discuss encapsulation here, although it actually plays a crucial role in the utility of ACL2 for large projects.

The form of an ACL2 function definition is as in Common Lisp,

`(defun` f `(`$x_1 ... x_n$`) (declare ...)` *body*`)`

ACL2 extends Common Lisp's `declare` so as to permit the specification of a guard expression, `(`$g\ x_1 ... x_n$`)`, as well as to permit the optional specification of an ordinal measure and other "hints." Some additional syntactic restrictions are put on *body*. These insure that the Common Lisp version of f will execute efficiently and in accordance with claims we make below. Roughly speaking, it is here that we enforce the syntactic notion that the current state is single threaded (by restricting the use of the variable named `state`) and insure that multiple values are used appropriately.

If the syntactic restrictions are met and the required termination theorems can be proved, then

Axiom.
`(`$f\ x_1\ ...\ x_n$`) =` *body*

is added as a new axiom. Observe that the axiom added is independent of the guard.

2.7 The Classic Example

Consider the definition of the function `app`. The predicate `(true-listp x)` requires that `x` be a "true list," i.e., a list ending in `nil`.

```
(defun app (x y)
  (declare (xargs :guard (true-listp x)))
  (if (null x)
      y
      (cons (car x) (app (cdr x) y))))
```

The termination condition requires that we exhibit a measure of the arguments that decreases in the recursion. Note that we recursively `cdr` the first argument, `x`, under the hypothesis that it is not `null`. The guard is ignored in the termination argument. A suitable measure (not shown) is akin to the length function but counts the length of `nil` to be 0 and the length of all other atoms to be 1. The recursion terminates because, logically speaking, in ACL2 `cdr` has been completed to return `nil` on non-conses.

After the function is admitted, the axiom

Axiom.
```
(app x y)
  =
(if (null x)
    y
    (cons (car x) (app (cdr x) y)))
```

is added. Note that the axiom does not mention the guard. Indeed we can prove the surprising theorem

Theorem. `pathological-app-call`
```
(equal (app 7 8) (cons nil 8))
```

We will return to this pathological example later.

We can also prove the very useful and powerful unconditional equality stating that `app` is associative.

Theorem. `associativity-of-app`
```
(equal (app (app a b) c) (app a (app b c)))
```

The proof takes advantage of the fact that `car` and `cdr` return `nil` on non–`cons` arguments such as numbers. Experience with Nqthm has shown that many such elegant theorems can be proved without need for "type-theoretic" hypotheses when reasonable "default" values are used for functions (e.g., `nil` for list-processing functions and `0` for numeric functions). Why is this elegance important, and why is it important that guards not be involved in the logical axioms added by definitions? We address the latter question first.

Had the guard of `app` been involved in the axiom defining the function, this unconditional equality would not be true. We would need to restrict `a` and `b` to being null-terminated lists. This would complicate the statement and proof of the theorem in several ways. First, in order to reason about the outermost call of `app` on the left hand side, we would have to show that `(app a b)` is a true list when `a` and `b` are true lists. Second, in an inductive argument we would have to show that the inductive instances chosen for the variables satisfy the guard in order to use the definition of `app`. Finally, when one proves an implication by induction extra cases arise from the propositional calculus (in order to detach the conclusion of the induction hypothesis).

Furthermore, once the restricted theorem were proved, we could use it only when we could prove the necessary conditions about the instances of `a` and `b`. The value of avoiding such peripheral reasoning is taught well by experience with Nqthm. The extra reasoning often requires additional theorems to be proved, slowing down the user, and is itself expensive when applying those theorems, slowing down the theorem prover.

Thus, there are many reasons to prefer the unrestricted form of the theorem and hence the form of the definition in which the guard plays no logical role. In fact, guards did play a logical role in earlier versions of ACL2, and we were driven to return to the Nqthm paradigm of total functions because of the complexity that guards introduced in some proof efforts.

3 The Relation Between ACL2 and Common Lisp

The implicit guards of Common Lisp allow great efficiency. There are implementations of Common Lisp, for example, Gnu Common Lisp, in which the performance of the compiled code generated for arithmetic and list processing functions can be comparable to hand-coded C arithmetic and pointer manipulation. Exceptional execution efficiency on a wide variety of platforms, combined with clear applicative semantics when used properly, was one of the great attractions of basing the ACL2 logic on Common Lisp.

Consider for example the primitive function `car`. Page 411 of [40] says that the argument to `car` "must be" a cons or `nil`. On page 6 we learn "In places where it is stated that so-and-so 'must' or 'must not' or 'may not' be the case, then it 'is an error' if the stated requirement is not met." On page 5 we learn that 'it is an error' means that "No valid Common Lisp program should cause this situation to occur" but that "If this situation occurs, the effects and results are completely undefined" and "No Common Lisp implementation is required to detect such an error."

Thus, an implementation of the function `car` may assume its actual is a cons or `nil`. By a suitable representation of data, the implementation of `car` can simply fetch the contents of the memory location at which the actual is stored. No type checks are necessary. Similarly, `app`, whose guard requires that the first argument be a list ending in `nil`, can recur down the `cdr` of that argument until it encounters `nil`, without type checking. Of course, if `car` or `app` is applied to `7` the results are unpredictable, possibly damaging to the runtime image, and usually implementation dependent. These aspects of Lisp make it difficult to debug compiled Lisp code.

This also raises problems with the direct embedding of applicative Common Lisp into a logic. The situation is far worse than merely not knowing the value of `(car 7)`. We do not know that the value is an object in the logic: `(car 7)` might be π, for example. Worse still, we do not know that `car` is a function: the form `(equal (car 7) (car 7))`, which is an instance of the axiom `(equal x x)`, might evaluate to `nil` in some Common Lisps because the first `(car 7)` might return 0 and the second might return 1.

ACL2 solves this problem by claiming a connection to Common Lisp only in situations in which functions are applied to their guarded domains. Furthermore, ACL2 provides a means of verifying that such situations obtain.

3.1 Gold Function Symbols and Terms

To make this precise, we define two inter-related notions: that of a function symbol being "gold"; and that of of a term being "gold" under some hypothesis. Roughly speaking, a function symbol is gold if, when its guard is true, the guards of all subroutines encountered during evaluation are true of their arguments. The following definition of both senses of *gold* ignores lambda expressions and mutually recursive function definitions because they can be eliminated by the definition of suitable auxiliary function symbols.

- All ACL2 logic primitive function symbols are *gold*.

- A defined function f with guard g and body b is *gold* if every function symbol mentioned in g is gold, the term g is gold under `t` (*true*), every function symbol besides f that is mentioned in b is gold, and the term b is gold under g.

- Variables and quoted constants are *gold* terms under all hypotheses.

- The term `(IF` a b c`)` is *gold* under h if a is gold under h, b is gold under `(AND` h a`)` and c is gold under `(AND` h `(NOT` a`))`.

- The term `(`f a_1 ... a_n`)`, where f is not `IF` and has guard `(`g v_1 ... v_n`)` (where the v_i are the formals of f) is gold under h provided each a_i is gold under h and `(IMPLIES` h `(`g a_1 ... a_n`))` is a theorem. The formula that must be proved is called the *guard conjecture* for the subterm in question.

When we say a term is gold without saying "under hypothesis h" we mean it is gold under the hypothesis `t`. We sometimes say a function or term is *Common Lisp compliant* as a synonym for saying it is gold. We call the process of checking whether a function symbol or term is gold *guard checking* or *guard verification*.

For example, consider again the definition of `app`.

```
(defun app (x y)
  (declare (xargs :guard (true-listp x)))
  (if (null x)
      y
      (cons (car x) (app (cdr x) y))))
```

The function symbol `app` is gold, i.e., Common Lisp compliant, because every previously defined function used in its definition is gold (they are all primitive) and when its body is evaluated, every guard encountered is true if the guard for `app` is true initially. The latter condition can be expanded as follows. There are three subroutines in the body of `app` that have non-trivial guards: `car`, `cdr`, and `app`. For `app` to be gold, a theorem must be proved for each call of such a subroutine in the body of `app`. The theorem requires that `(true-listp x)` (the initial guard of `app`), together with the tests leading to the call in question, imply the guard of the call. These three theorems are trivial to prove.

Consider the "unguarded" version of `app`, say `ug-app` defined analogously to `app` but with a guard of `t`. That function symbol is not gold. For example, the guard conjecture for the `(car x)` subexpression is invalid: the negation of the `(null x)` test by itself is insufficient to insure the guard for `car`. Nevertheless, `ug-app` is admissible in ACL2, it is just not Common Lisp compliant. Note further that the functions denoted by `ug-app` and `app` can be proved equal in the logic, since the guard is not part of the defining axiom. Thus, the notion of being gold is a syntactic property of a function symbol having to do with the form of its definition.

Guards and the notion of Common Lisp compliance can have a dramatic impact on the efficiency with which an ACL2 expression can be evaluated. We return to this point in a moment.

3.2 The Story Relating the Logic to Common Lisp

We claim that if a function symbol of ACL2 is gold and a gold theorem has been proved about it, then every execution of that function in any compliant Common Lisp produces answers consistent with the theorem, provided the arguments to the function satisfy the guard and no resource errors (e.g., stack overflow) occur. Less precisely, gold ACL2 theorems describe the behavior of Common Lisp.

Consider `app` again. It is a gold function symbol. We have the following theorem, which is easily proved by using the fact that both `car` and `cdr` return `nil` on non–`cons` arguments such as numbers.

Theorem. `pathological-app-call`
```
(equal (app 7 8) (cons nil 8))
```

Does our claim mean that in every compliant Common Lisp, `(app 7 8)` evaluates to `(cons nil 8)`? No! The reason is that the theorem is not gold. The guard for `app` is violated by `7`. Nothing can be inferred about Common Lisp via our claim. Some Common Lisps may cause severe trouble if commanded to evaluate `(app 7 8)`.

How about the associativity result for `app`?

Theorem. `associativity-of-app`
```
(equal (app (app a b) c) (app a (app b c)))
```

Can we expect `app` to be unconditionally associative in Common Lisp? Again, the answer is no because the theorem is not gold.

However, the following theorem is gold. (It is stated in the form `(if x y t)` rather than `(implies x y)` because the definition of "gold" gives special treatment to `if` but not to `implies`.)

Theorem. `gold-associativity-of-app`
```
(if (and (true-listp a)
         (true-listp b))
    (equal (app (app a b) c) (app a (app b c)))
    t)
```

Of course, we must prove this theorem, but its proof is trivial given the unconditional associativity result!

We must also verify that it is gold. This is somewhat more interesting. The guard condition for `(app (app a b) c)` is the beautiful theorem `(implies (and (true-listp a) (true-listp b)) (true-listp (app a b)))`.

Our claim that gold theorems describe Common Lisp can be made more precise as follows. We present the claim in a restrictive setting here for simplicity. Suppose f is a function symbol of one argument defined in some *certified book* (e.g., a file of admissible ACL2 definitions and theorems), that the guard of f is `t`, that f is gold, and that `(equal (`f x`) t)` is a (necessarily gold) theorem of ACL2 proved in that book. Consider any Common Lisp compliant to [40] into which the ACL2 kernel has been loaded. `Load` the book into that Lisp. Let x be a Common Lisp object that is also an object of ACL2. Then the application in that Lisp of f to x returns `t` or else causes a resource error (e.g., stack overflow or memory exhaustion).

The essence of the proof of this claim is to observe that `(`f x`)` evaluates to `t` in the logic (because of the soundness of the logic), and the computation will at no step exercise a function symbol outside of its guarded domain (because f is gold). Since the logic and Common Lisp agree inside the guarded domain, the Common Lisp computation of `(`f x`)` returns `t` also.

A less restrictive alternative formulation is that if *thm* is a gold theorem in some certified book then any ACL2 instance of *thm* evaluates to non-`nil` in any compliant Common Lisp into which the ACL2 kernel and the book have been loaded.

3.3 Guards and Efficiency

One obvious implication of the "Story" is that if one has a formal model that has been proved Common Lisp compliant and one wishes to evaluate gold applications of the model, one can ignore the ACL2 theorem prover altogether, load the model into a compliant Common Lisp (containing the ACL2 kernel), and directly execute the model to obtain results consistent with the axioms. For example, one might build a gold simulator of a microprocessor or high-level language and provide it to users via a stand-alone Common Lisp engine (perhaps with some pleasant interface). The point is that the ACL2 theorem prover and interface need not be present.

A less obvious use of our claim is made in the ACL2 system itself. Consider the variable-free term `(app 7 8)`. Suppose we wish to determine the value of this expression under the axioms. This might happen if the ACL2 user wishes to test the proposed definition of `app` or it might happen when the term arises within a proof, e.g., by instantiation of some previously proved lemma. Can we evaluate the term in Common Lisp? No. The term is not gold and so our claim relating the logic to Common Lisp tells us nothing. In order to evaluate this term, the ACL2 system contains a special interpreter that gives the axiomatic interpretation to function symbols outside their guarded domains. For example, that interpreter must evaluate `(car 7)` to `nil`. This is slow because the interpreter must check all guards at runtime. (In fact this "interpreter" is implemented by introducing auxiliary Common Lisp functions corresponding to the function symbols, and they can be compiled. But still, those functions are responsible for checking guards at runtime.) This style of evaluation is the only one provided by Nqthm.

Now consider the variable-free term `(app '(1 2 3 4 5) '(6 7 8))`. This term is gold because `app` is gold and the arguments satisfy the guard of `app`. Thus, to obtain the value of this term under the axioms we can simply evaluate the term in Common Lisp. This is considerably more efficient than interpreting it with the ACL2 interpreter, not just because the Common Lisp definition of `app` can be compiled but because the compiled code need check no guards at runtime.

The ACL2 system is engineered to so use the underlying Common Lisp engine. Thus, one important incremental effect of proving that an ACL2 function symbol is gold is that subsequent applications of the function are more efficiently computed.

3.4 Guards as a Specification Device

Guards may also be used as type specifications. Gold functions are "well-typed." However, guards are much more expressive than conventional types, because they can be arbitrary terms in the logic. Of course, ACL2 "type checking" is not decidable for this same reason. For some related work, see [1].

If one attaches restrictive guards to one's functions and then proves the functions are gold, one obtains assurance that the functions are being exercised only on their intended domains. More precisely, one gains the knowledge that the computed value is provably equal to the function application in a weakened logical system in which the equality of each function application to its body is conditional on its guard being true. Nqthm provides no such assurance.

3.5 Separation of Concerns

Note how ACL2's treatment of guards separates concerns. For theorem proving simplicity in the Nqthm tradition ACL2 makes all functions total by completing the primitives with arbitrary but "natural" default values. Functions can be introduced into the logic without addressing the question of whether they are compliant with Common Lisp. Their properties can be proved without concerning oneself with questions of whether guards are satisfied. Often this allows the properties themselves to be more simply stated. This allows the data base of rules to be less restrictive, more powerful, and more easily applied. Nevertheless, logical functions can be evaluated with Nqthm efficiency.

Once a system of ACL2 functions has been defined and its logical properties proved, one can move on to the question of Common Lisp compliance, either to gain execution efficiency in an ACL2 setting or in a stand-alone Common Lisp, or to gain type assurance. Efficiency can be gained incrementally by doing guard verification on the core subroutines but not on the outlying checkers, preprocessors and postprocessors typically involved in a big system. Furthermore, having proved certain functions gold one can stop and settle for the corresponding efficiency or type assurance or one can prove that the key properties proved are also gold.

Recall for example that one can carry out the following sequence of steps:

- admit `app` as a function,
- prove it unconditionally associative,
- prove it gold, i.e., Common Lisp compliant or well-typed,
- trivially prove a restricted version of its associativity, and then
- prove that restricted version of associativity to be gold or Common Lisp compliant.

In versions of ACL2 predating Version 1.8, where guards were part of the definitional equations, these issues were often intertwined so that it was impossible to address them separately. While this makes little difference in a setting as simple as `app` and its associativity it makes a great deal of difference in models involving thousands of functions and properties.

4 Concluding Observations: Notions of "Industrial Strength"

We have argued that ACL2 is of "industrial strength." Up to now our main argument has been its improved efficiency over Nqthm by virtue of being executable as Common Lisp, with special consideration for efficient execution of operations involving arrays, property lists, and state. We have also indicated that a very expressive kind of type-correctness can be gained by "guard" verification, and yet this capability is separated from the logic proper so that proofs are not needlessly hindered. Below we consider some strengths of ACL2 besides efficiency as a programming language: robustness, general features, maintainability, and proof support.

A notion of "industrial strength" is the robustness of the tool. We have put considerable effort into making the program bullet-proof, handling user errors graciously and with appropriate messages. The interface is consistent, providing the ability to submit definitions and theorems as well as the ability to execute applicative Lisp code efficiently.

Yet another notion of the "industrial strength" of a tool is its support for features that are crucial to get the job done. Here is a partial list of such features offered by ACL2.

- Extensible on-line documentation that may be read at the terminal, as well as through text and by way of hypertext (Emacs Info, HTML)
- Support for undoing back through a given command as well as "undoing an undo"
- A notion of "books" that allows independent development and inclusion of libraries of definitions and theorems, with a way to apply `local` to forms that should not be exported from a given book
- A "program" mode that allows definition and execution of functions without any proof burden being imposed, and without any risk of unsoundness being introduced (as the prover does not know about "program" mode functions)
- A "realistic" collection of data types that includes strings and (complex) rational numbers, with support for reasoning about such data (e.g., a fully integrated linear arithmetic decision procedure for the rationals)

- Extensive capabilities for controlling the prover (see below)
- Common Lisp macros for ease of programming and specification without cluttering up the collection of functions about which one needs to reason
- Many useful programming primitives, including primitives for efficient passing of multiple values and for I/O
- Common Lisp *packages* that support distinct namespaces

We also have found that the applicative style of programming is amenable to maintenance, both for fixing bugs and for implementing enhancements. Moreover, the subset of Common Lisp that ACL2 supports has been sufficient to code all but the very lowest levels of the system (which are needed to implement some of the primitives). We believe that a Common Lisp program that is applicative and perhaps even performs I/O is in fact an ACL2 program, or very nearly so.

As with Nqthm and Pc-Nqthm, proofs of significant theorems in ACL2 tend to require a serious effort on the part of the user to prove appropriate supporting lemmas, primarily stored and used as (conditional) rewrite rules. However, ACL2 offers considerably more support for proofs than those, or indeed (we believe) other current general-purpose theorem provers, including the following:

- As with Nqthm, a proof commentary in English that assists users in debugging failed proofs
- As with Pc-Nqthm, an interactive loop for proof discovery that is extensible through "macros" and has access to the full power of the theorem prover
- The capability to apply hints to individual subgoals
- "Proof tree" displays to show the evolving structure of the proof in real time, and which also make it convenient to inspect failed proofs
- Efficient handling of propositional logic, normally through a clause generator that is much more efficient than Nqthm's, but also through a facility that integrates ordered binary decision diagrams with rewriting
- A "functional instantiation" facility that gives ACL2 (like Nqthm) some of the convenience of a higher-order logic without sacrificing the simplicity of a first order logic.
- A "break-rewrite" facility, more sophisticated than that in Nqthm (or any other prover as far as we know), that allows proof-time debugging of the rewrite stack
- A "theory" mechanism that makes it easy to manipulate sets of rules, especially turning them on and off but also checking desired invariants on sets of rules

- A "forcing" mechanism that gives the prover permission to defer checking of hypotheses of specified rules until the end of the "main" proof
- Support for a variety of types of rules (17), including types supported by Nqthm. These include:
  - Conditional rewrite rules, which may be used not only to replace equals by equals but may also work with respect to user-defined equivalence (congruence) relations
  - Linear arithmetic rules, together with a mechanism to create rules for certain simple orders other than the standard "less-than" order on the rationals
  - "Forward chaining," "type prescription," "compound recognizer," and "built in clause" rules for efficient automatic use of certain facts
  - An improved "meta lemma" facility that allows such lemmas to be conditional (i.e., have hypotheses)
  - Rules for use outside the simplifier/rewriter: elimination and generalization

Thus, we claim that ACL2 is industrial strength in its efficiency, its consistent and robust interface, its array of general features, its ease of maintenance, and the flexibility of its theorem prover.

Of course, ultimately the test for whether a tool is "industrial strength" must be whether it can be used to do jobs of interest to industry. The first two important applications of ACL2 support our claims that it is up to the task:

- In collaboration with Motorola, Inc., we have produced an executable formal specification of the Motorola CAP [17], a digital signal processor designed by Motorola to execute a 1024 point complex FFT in 131 microseconds. Every well-defined behavior of the CAP is modeled, including the pipeline, I/O, interrupts, breakpoints and traps (but excluding the hard and soft reset sequences). The CAP is much more complex than other processors recently subjected to formal modeling, namely the FM90001, MC68020, and AMP5. In principle, a CAP single instruction can simultaneously modify well over 100 registers. Nevertheless, ACL2 can compute the symbolic effects of a complicated instruction in just a few seconds and is a faster code debugging tool than a conventional engineering simulation model. With ACL2 we have proved that under suitable conditions our model of the CAP is equivalent to a simpler pipeline-free model. We have used ACL2 to verify several CAP assembly code application programs, including an FIR filter and a peak finding algorithm that uses the adder array as a chain of comparators.

- In collaboration with Advanced Mirco Devices, Inc., we have formalized and mechanically proved the correctness of the kernel of the floating point division algorithm used on the AMD5_K86^{TM} microprocessor, AMD's first Pentium-class processor. In particular, in [30] we prove that if p and d are 64,,15 (possibly denormal) floating point numbers, $d \neq 0$ and $mode$ specifies one of six rounding procedures and a desired precision $0 < n \leq 64$, then the output of the algorithm is p/d rounded according to $mode$. We also prove that every intermediate result is a floating point number in the format required by the resources allocated to it.

ACL2 is thus being used to tackle problems of importance to industry.

References

[1] R. L. Akers. Strong Static Type Checking for Functional Common Lisp. Ph.D. Thesis, University of Texas at Austin, 1993. Also available as Technical Report 96, Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703. URL http://www.cli.com/.

[2] K. Albin. 68020 Model Validation Testing, CLI Note 280, August 1993.

[3] W. R. Bevier. A Verified Operating System Kernel. Ph.D. Thesis, University of Texas at Austin, 1987. Also available as Technical Report 11, Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703. URL http://www.cli.com/.

[4] W. R. Bevier, W. A. Hunt, J S. Moore, and W.D. Young. Special Issue on System Verification. *Journal of Automated Reasoning*, **5**(4), 409–530, 1989.

[5] W. R. Bevier and W. D. Young. Machine Checked Proofs of the Design of a Fault-Tolerant Circuit, *Formal Aspects of Computing*, Vol. 4, pp. 755–775, 1992. Also available as Technical Report 62, Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703, August, 1990 (URL http://www.cli.com/), and as NASA CR-182099, November, 1990.

[6] R. S. Boyer, D. Goldschlag, M. Kaufmann, and J S. Moore. Functional Instantiation in First Order Logic. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press, 1991, pp. 7-26.

[7] Robert S. Boyer, Matt Kaufmann, and J Strother Moore. The Boyer-Moore Theorem Prover and Its Interactive Enhancement. *Computers and Mathematics with Applications*, bf 29(2), 1995, pp. 27-62.

[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 R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*, Academic Press, 1981.

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

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

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

[13] R. S. Boyer and Y. Yu, Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor. In D. Kapur, editor, *Automated Deduction – CADE-11, Lecture Notes in Computer Science 607*, Springer-Verlag, 416–430, 1992.

[14] B. C. Brock, W. A. Hunt, Jr. and W. D. Young. Introduction to a Formally Defined Hardware Description Language. In *Theorem Provers in Circuit Designs*, Number A10 in IFIP Transactions. North Holland, 1992.

[15] J. Cowles. Meeting a Challenge of Knuth. Internal Note 286, Computational Logic, Inc., Austin, Texas, September, 1993.

[16] D. Craigen, S. Kromodimoeljo, I. Meisels, B. Pase, M. Saaltink. EVES: An Overview. Odyssey Research Center, ORA Conference Paper CP-91-5402-43, March, 1991.

[17] S. Gilfeather, J. Gehman, and C. Harrison Architecture of a Complex Arithmetic Processor for Communication Signal Processsing in *SPIE Proceedings, International Symposium on Optics, Imaging, and Instrumentation*, **2296** *Advanced Signal Processing: Algorithms, Achitectures, and Implementations V*, 624–625, July, 1994.

[18] D. M. Goldschlag. Mechanizing Unity. In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*, North Holland, Amsterdam, 1990.

[19] D. M. Goldschlag. Mechanically Verifying Concurrent Programs with the Boyer-Moore Prover. *IEEE Transactions on Software Engineering*, 16, September, 1990.

[20] W. A. Hunt, Jr. and B. Brock. A Formal HDL and its use in the FM9001 Verification. *Proceedings of the Royal Society*, 1992.

[21] M. Kaufmann. A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover, Technical report 19, Computational Logic, Inc., May, 1988. URL http://www.cli.com/.

[22] M. Kaufmann. Addition of Free Variables to an Interactive Enchancement of the Boyer-Moore Theorem Prover, Technical Report 42, Computational Logic, Inc., 1990. URL http://www.cli.com/.

[23] M. Kaufmann. An extension of the Boyer-Moore theorem prover to support first-order quantification. *Journal of Automated Reasoning*, **9**(3):355–372, December 1992.

[24] M. Kaufmann. An Assistant for Reading Nqthm Proof Output. Technical Report 85, Computational Logic, Inc., November, 1992. URL http://www.cli.com/.

[25] M. Kaufmann and P. Pecchiari. Interaction with the Boyer-Moore Theorem Prover: A Tutorial Study Using the Arithmetic-Geometric Mean Theorem, Technical Report 102, Computational Logic, Inc., 1994. Also: To appear in the *Journal of Automated Reasoning*.

[26] M. Kaufmann and J S. Moore, High-Level Correctness of ACL2: A Story (DRAFT), URL ftp://ftp.cli.com/pub/acl2/v1-8/acl2-sources/reports/story.txt, September, 1995.

[27] M. Kaufmann and J S. Moore, ACL2 Version 1.8, URL ftp://ftp.cli-.com/pub/acl2/v1-8/acl2-sources/doc/HTML/acl2-doc.html, September, 1995.

[28] D. E. Knuth. Textbook examples of recursion. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press, San Diego, CA, 207–229, 1991.

[29] K. Kunen. A Ramsey Theorem in Boyer-Moore Logic, *Journal of Automated Reasoning*, **15**(2) October, 1995.

[30] J S. Moore, T. Lynch, and M. Kaufmann, A Mechanically Checked Proof of the Correctness of the Kernel of the AMD5$_K$86 Floating Point Division Algorithm, March, 1996 (submitted). URL http://devil.ece.utexas.edu:80/~lynch/divide/divide.html.

[31] J S. Moore. Verified Hardware Implementing an 8-Bit Parallel I\O Byzantine Agreement Processor. Technical Report 69, Computational Logic, Inc., Austin, Texas, August, 1991.

[32] J S. Moore. A Formal Model of Asynchronous Communication and Its Use in Mechanically Verifying a Biphase Mark Protocol, *Formal Aspects of Computing* **6**(1), 60–91, 1994.

[33] J S. Moore. Introduction to the OBDD Algorithm for the ATP Community, *Journal of Automated Reasoning* 12, 33–45, 1994.

[34] K. M. Pitman *et al. draft proposed American National Standard for Information Systems — Programming Language — Common Lisp; X3J13/93-102.* Global Engineering Documents, Inc., 1994.

[35] D. M. Russinoff. A Mechanical Proof of Quadratic Reciprocity. *Journal of Automated Reasoning*, **8**(1), 3–21, 1992.

[36] D. M. Russinoff. Specification and Verification of Gate-Level VHDL Models of Synchronous and Asynchronous Circuits. Technical Report 99, Computational Logic, Inc., Austin, Texas, May, 1994. URL http://www.cli.com/.

[37] N. Shankar. A Mechanical Proof of the Church-Rosser Theorem. *JACM* **35**(3), 475–522, 1988.

[38] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*, Cambridge University press, 1994.

[39] G. L. Steele Jr. *Common LISP: The Language*, Digital Press: Bedford, MA, 1984.

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

[41] M. Wilding. An optimal real-time scheduler based on a simple model of computation. Internal Note 276, Computational Logic, Inc., July 1993.

[42] W.D. Young. Verifying the Interactive Convergence Clock Synchronization Algorithm using the Boyer-Moore Theorem Prover. Contractor Report 189649, NASA, April 1992. Also available as Technical Report 77, Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703. URL http://www.cli.com/.

[43] Y. Yu. Automated Proofs of Object Code for a Widely Used Microprocessor. Ph.D. Thesis, The University of Texas at Austin, 1992. Also available as Technical Report 92, Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703 and through Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, California, 94301.