Towards a Mechanically Checked Theory of Computation: A Progress Report

J Strother Moore*

May 21, 1999

Abstract

Formal mathematical logic is ideally suited to describing computational processes. We discuss the use of a mechanized mathematical logic, namely ACL2 (A Computational Logic for Applicative Common Lisp) to model computational problems and to prove theorems about such models.

1 Prelude

In 1961, John McCarthy first presented the seminal paper "A Basis for a Mathematical Theory of Computation" [23]. In that paper McCarthy defined the class of computable functions on some set of base functions. He discussed other fundamental issues, such as the role of non-computable functions, quantification, functionals and what we now call abstract data types. He introduced the notion of "recursion induction." He used his formal system to prove many now classic elementary theorems in his emerging theory of computation, including distributivity of Peano multiplication over Peano addition and the associativity of the list concatenation function. He also clearly laid the basis for the formal establishment that two computational processes are equivalent or that they stand in some other precisely defined relationship.

In a companion paper written a year later [22], McCarthy wrote

> Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer program. For this to be possible, formal systems are required in which it is easy to write proofs. There is a good prospect of doing this, because we can require the computer to do much more work in checking each step than a human is willing to do. Therefore, the steps can be bigger than with present formal systems.

In the three and a half decades since these papers were published, McCarthy's vision has inspired many of us. In 1971, Bob Boyer and I wrote a Lisp program, called the Pure Lisp Theorem Prover [1], capable of automatically finding many of the example proofs exhibited in [23]. That early computer program evolved over the next decade to become the Boyer-Moore theorem prover, NQTHM [2, 3]. Then Matt Kaufmann and I, with the advice of Boyer, collaborated to produce ACL2 [16, 17].

While still undeniably a research tool, ACL2 finds commercial applications in the mechanical checking of formally stated properties of microprocessors, microcode, and other such artifacts. While not nearly as ambitious as McCarthy's 1961 vision of a theory of computation, ACL2 is a reification of it. In this paper I discuss ACL2 and its use as a mechanized theory of computation.

*Department of Computer Sciences, University of Texas, Austin, TX 78712, moore@cs.utexas.edu.

2 Introduction

Microprocessors are changing our world. They are becoming ubiquitous, popping up in places both expected (e.g., computers, airplanes, missiles, televisions, automobiles) and unexpected (e.g., toothbrushes, bicycles, light switches, and greeting cards).

In a very important sense, microprocessors are just state machines: at every tick of the clock the machine changes from one state to the next according to a precisely specified mathematical "state transition" function. States are discrete. They are typically composed of "atomic" objects such as integers and words, glued together into more complex structures such as sequences, tables, trees, etc. For the simplest machines, the state transition function takes just one argument, the current state, and produces the next state. In general, the function may take additional arguments representing some kind of "input" impinging on the machine "from outside" and may produce, in addition to a new state, some "output." For our purposes, machines may have either a finite or an infinite number of distinct states.

State machines are well-studied mathematical abstractions. There are many different kinds: finite state automata, Moore machines, Mealy machines, Turing machines, etc. Such machines may be described formally by describing the legal states and the state transition function. Such descriptions are typically written in some logical language. Indeed, mathematical logic, especially a logic supporting recursive definition and inductive construction, is almost ideally suited to describing state machines formally. While simple machines might be described by relatively simple languages, as used in automata theory, the state machines implemented by microprocessors can, for all practical purposes, only be described in a general-purpose mathematical logic.

This places us at an extremely interesting moment in the history of mathematical logic: applications of great practical importance are plentiful. Most of mathematics has long enjoyed – or suffered from – this convergence of theory and practice; but until the invention of the digital computer, mathematical logic was almost exclusively a tool for the study of logic itself or of philosophy. Logic was too "black and white" to describe the "real world" accurately. This, of course, motivated many interesting developments in logic, e.g., non-monotonic logic and fuzzy logic. But classical logic remained a tool without a practical application. The complexity and ubiquity of microprocessors has changed that. Microprocessors are *intended* to be discrete state machines. They realize this intention so faithfully and so directly that mathematical logic is the language of choice for writing down their descriptions. Indeed, until one descends in the design hierarchy to the point at which gates and wires are laid out, microprocessors are so accurately described by logical formulas that there is often confusion over whether by "microprocessor" one means the mathematical abstraction or the physical artifact. In this paper, I take the view that a microprocessor is a physical artifact that is *modeled* by a set of logical formulas. But quite often in microprocessor design there are levels of abstraction above that low-level model, i.e., at the instruction-set level, at which there is no distinction between the machine and the set of formulas describing it: the artifact of interest *is* the abstract state machine, not some imperfectly realized physical object.

That microprocessors can be accurately described by a classical formal logic is not the end of the story. The complexity of today's microprocessor designs is such that mathematical proof is the only way to gain confidence that a design enjoys some desired property.

Therefore, there is a growing community of applied logicians: people trained in creating formal models of microprocessors and producing formal proofs of theorems about those models. Typical models consist of many definitions of "new" concepts, such as "the registers upon which instruction *ins* depends" and whether "processor i has the exclusive write permission for address a." These concepts are combined to produce descriptions of machines and their desired properties. Machine models can require hundreds of pages to write down fully.

Proving theorems about these models can be a daunting task. For this reason, many applied logicians prefer to use mechanized tools to develop and/or check their proofs. ACL2 is one such tool.

3 The ACL2 Logic

"ACL2" is both the name of a mathematical logic and the mechanical theorem proving system supporting it. It stands for A Computational Logic for Applicative Common Lisp. The ACL2 logic is a first-order essentially quantifier-free logic of recursive functions and inductively defined objects, developed by Matt Kaufmann and the author. It is modeled closely on the Boyer-Moore NQTHM logic [3] but describes an applicative subset of Common Lisp [10] rather than a "home-grown" Lisp.

Our style of formalizing ACL2 logically follows that of Shoenfield [32]. We introduce quantifier-free first order logic with equality between terms. The syntax of terms is that of Common Lisp. We then add axioms describing five Common Lisp data types: complex rationals (with the rationals and the integers as subsets), characters, strings, symbols (with packages), and conses. We add axioms defining about 170 Common Lisp functions, including such basic list processing functions as `if`, `equal`, `cons`, `car`, `cdr`, `consp`, and `atom`, as well as the primitives for manipulating the other data types, such as `+`, `-`, `*`, and `/`; `code-char` and `char-code` (mapping between naturals and characters); `coerce` (mapping between lists of characters and strings); and `intern` and `symbol-name` (mapping between strings and symbols). We define the function `acl2-count` to be a measure of the size of an object. The size of a `cons`-tree is the number of `cons` nodes in it plus the sizes of the leaves.

Following Goodstein [11], we identify a certain set of objects, constructed from conses and natural numbers, with the ordinals below $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$. These objects are recognized by the function `e0-ordinalp`. We define `e0-ord-<` to test when one ordinal is less than another. In a suitable set theory one could prove that `e0-ord-<` is well-founded. This theorem cannot be stated in the weak logic of ACL2, but its validity is fundamental to the soundness of the formal logic.

We allow the definitional extension of the logic by the addition of new recursion equations, under certain syntactic conditions, provided certain syntactically specified "measure conjectures" can be proved. These conjectures establish that some ordinal measure of the arguments to the function decreases according to `e0-ord-<` in the recursive calls. Consider

```
(defun append (x y)
  (if (consp x)
      (cons (car x) (append (cdr x) y))
      y)).
```

This definition is admitted using the measure `(acl2-count x)`. The measure conjectures, `(e0-ordinalp (acl2-count x))` and `(implies (consp x) (e0-ord-< (acl2-count (cdr x)) (acl2-count x)))`, are theorems.

Only terminating recursions are allowed under the definitional principle. This preserves the soundness of the logic, which otherwise would be imperiled by "definitions" such as `(defun f (x) (+ 1 (f x)))`. In our simple first-order logical setting, nonterminating "definitions" can give rise not just to "ambiguous" functions (to use the terminology of [23]) but to inconsistency.

We also allow inductions up to ϵ_0 in a manner exactly dual to the definitional principle. The definitional principle and the induction rule of inference are taken virtually unchanged from NQTHM [3].

Using the "encapsulation" mechanism, the logic may be extended by the introduction of undefined function symbols axiomatized to satisfy certain constraints. However, theorems must be proved establishing that such functions exist. A derived rule of inference, called "functional instantiation," allows theorems about constrained functions to be instantiated in a second-order way with function symbols known to satisfy the constraints. See [6, 18].

Finally, we add several other logical features, including the ability to introduce a function which returns an unspecified object satisfying a given formula, provided such an object exists. See `defchoose` in [17]. This is used to provide the power of full first-order quantification; see `defun-sk` in [17].

A subset of ACL2 is executable in the sense that some ground expressions may be reduced to primitive constants by the systematic use of instantiation and substitution of equals for equals. (The subset excludes constrained functions.) Furthermore, ACL2 is consistent with Common Lisp in that ground instances of certain expressions may be computed by simply evaluating the instances in any compliant Common Lisp (subject to resource limitation). To make this precise we introduce the notion of the "intended domain" of Common Lisp functions. The Common Lisp standard [10] does not specify the behavior of functions outside their intended domains (but ACL2 does). We provide a means by which one can establish that an expression's evaluation stays within the intended domains of the functions involved. This is the so-called "guard" mechanism of [16]. If a function's "guard conjectures" are theorems, appropriate calls of the function can be evaluated by direct appeal to Common Lisp.

4 The ACL2 Theorem Prover

The ACL2 theorem prover was also inspired by NQTHM's [2] but has been considerably improved. Like NQTHM, ACL2 supports fast type reasoning, complete propositional calculus and equality, a built-in linear arithmetic decision procedure, backwards chaining, conditional rewriting, verified meta-theoretic simplifiers, a variety of heuristics designed to generalize a conjecture before resorting to mathematical induction, and sophisticated heuristics for inventing often-appropriate inductive schemas. All of these proof techniques are closely integrated and driven by a database of previously proved theorems. By the careful selection of lemmas to prove first, the informed user of either NQTHM or ACL2 can "program" the system to find a given proof.

In addition, ACL2 has integrated many other proof techniques, including faster propositional calculus, the mixing of BDDs and rewriting ([24] and the `BDD` hint in [17]), forward chaining, many devices for controlling the application of previously proved lemmas, including meta-theoretic simplifiers, and, perhaps most importantly, the implementation of congruence-based rewriting (see **`congruence`** in [17]). The last permits the user to introduce new equivalence relations and cause ACL2 to use theorems about those relations as rewrite rules in appropriate contexts defined by congruences established by other lemmas.

This is especially useful in a logic such as ours where abstract objects are represented concretely. Consider finite sets represented as lists and let **`(union a b)`** be defined to be the union operator. Because of duplication and ordering, it is not in general the case that **`(union a b)`** is equal to **`(union b a)`**. Therefore, **`union`** expressions cannot in general be commuted. One can define the relation **`set-equal`**, testing whether two lists have the same elements while ignoring duplications and order. One can then prove **`(set-equal (union a b) (union b a))`**. However, this lemma is not very useful, by itself, because it cannot be used as a rewrite rule to simplify expressions: it is not an equality! But ACL2 takes special notice of such rules if **`set-equal`** has been proved to be an equivalence relation. Furthermore, congruence lemmas can be proved, establishing, for example, that **`set-equal`** is a congruence (modulo **`iff`**) for **`mem`**, the set-membership relation: **`(implies (set-equal x y) (iff (mem e x) (mem e y)))`**. The above theorem, which may be abbreviated as simply **`(defcong perm iff (mem e x) 2)`**, informs the ACL2 rewriter as follows. Suppose the rewriter encounters an expression of the form **`(mem`** δ **`(union`** α β**`))`** in a context in which the rewriter is trying to maintain propositional equivalence (**`iff`**). That is, suppose the **`mem`** expression occurs as a hypothesis or conclusion of a formula to be proved, or as the test of an **`if`**. The congruence lemma tells us that the propositional value of the **`mem`** expression is unchanged even if we replace the second argument by some **`set-equal`** value. Therefore, when rewriting the second argument of the **`mem`** expression, **`(union`** α β**`)`**, we may use use **`(set-equal (union a b) (union b a))`** as a rewrite rule, as though it were **`(union a b)`** $=$ **`(union b a)`**. (Heuristics determine whether ACL2 will use such a rule to permute the arguments of a function.)

Readers wishing more background on the mechanization of ACL2 are urged to see [2, 3], where the earlier NQTHM is described, plus the extensive ACL2 user's manual. The last is available in hypertext format on the internet, see [17].

5 Elementary Examples

We now turn to the main topic of this paper, which is the demonstration of ACL2 as a mechanized theory of computation. We do not show ACL2's proofs, but rather what the user must say to ACL2 to lead it to those proofs. We start with several examples from [23]. Our basic position is that a mechanized logic of computation ought allow the user simply to observe that elementary formulas are theorems; the underlying theorem prover should do the computation necessary to fill in the proof. The presentation below contrasts McCarthy's observations with ACL2's ability to follow them. Unless otherwise indicated, ACL2 fills in the proofs without help.

McCarthy defines the list concatenation function as the binary operator "*" with the notation:

$$x * y = [null[x] \to y; \mathtt{T} \to cons[car[x]; cdr[x] * y]].$$

Since "*" is already defined in our system (to be multiplication), we change the name to "`jmc::*`," i.e., to the symbol with the name `"*"` in a new package named `"jmc"`. This symbol can be written simply as `*` if we select `"jmc"` the "current package." By importing into that package the other Lisp symbols we need, we can then write:

```
(defun * (x y)
  (if (null x)
      y
      (cons (car x) (* (cdr x) y))))
```

As written above, this definition is not admitted by ACL2. The reason is that no measure is supplied and so the system guesses the measure `(acl2-count x)`. But that measure does not decrease. For example, if `x` is `0`, then `(null x)` is not true and hence the definition recurs to `(cdr 0)`. In ACL2, `cdr` returns `nil` when applied to non-`cons` objects. But both `0` and `nil` are of size `0`, as measured by `acl2-count`. Hence the guessed measure does not decrease. (McCarthy deals with problem by implicitly limiting the function to lists that terminate in `nil`.)

ACL2 can admit the function above, but the user must supply a measure that gives `nil` a smaller size than all other objects.

```
(defun * (x y)
  (declare (xargs :measure (if (null x) 0 (+ 1 (acl2-count x)))))
  (if (null x)
      y
      (cons (car x) (* (cdr x) y))))
```

With this measure, the definition is admitted. Note that `(* '(1 2 3) '(4 5 6))` is `'(1 2 3 4 5 6)`, as expected. Perhaps unexpectedly, `(* 0 '(4 5 6))` is `'(nil 4 5 6)`, because `0` is not `null` and has `car` and `cdr nil`.[1]

McCarthy next proves $[x * y] * z = x * [y * z]$ by recursion induction. ACL2 proves this without help from the user, in response to the command

```
(defthm associativity-of-*
  (equal (* (* x y) z) (* x (* y z))))
```

[1]Note the difference between this `*` function and the earlier shown `append`: The two agree when the first argument ends in `nil`, e.g., `(append '(1 2 3) '(4 5 6))` is `'(1 2 3 4 5 6)`. But they disagree if the first argument ends in a non-`nil` atom, e.g., `(append 0 '(4 5 6))` is `'(4 5 6)` rather than `'(nil 4 5 6)`. We generally use `append` for concatenation, preferring to treat all non-`consp` objects as "terminal markers."

The proof is by induction on **x**. When the proof is completed, ACL2 builds this lemma into its database of rewrite rules. Henceforth, the simplifier will right-associate *****-nests.

McCarthy then proves $\mathtt{NIL} * x = x$ and $x * \mathtt{NIL} = x$, the latter by induction. In ACL2, the former is trivially proved and the latter is not valid (consider $x = 0$). But the following is a theorem

```
(defthm *-left-id
  (implies (true-listp x)
           (equal (* x nil) x)))
```

which ACL2 can prove without assistance from the user. The predicate **(true-listp x)** is true if **x** is a list terminated by **nil**. The proof of ***-left-id** is by induction on x.

Next, McCarthy defines

$$reverse[x] = [null[x] \to \mathtt{NIL}; \mathtt{T} \to reverse[cdr[x]] * cons[car[x]; \mathtt{NIL}]].$$

The analogous

```
(defun reverse (x)
  (declare (xargs :measure (if (null x) 0 (+ 1 (acl2-count x)))))
  (if (null x)
      nil
      (* (reverse (cdr x)) (cons (car x) nil))))
```

is easily admitted by ACL2, but note that the user must supply the same special measure explaining this recursive scheme.

Finally, McCarthy states without proof two theorems which are "not difficult to prove by recursion induction."

$$reverse[x * y] = reverse[y] * reverse[x],$$

and

$$reverse[reverse[x]] = x.$$

The McCarthy-style proof of the first is by recursion induction with a "base case" of $x = \mathtt{NIL}$. The conjecture simplifies to the goal $reverse[y] = reverse[y] * \mathtt{NIL}$, which is proved by the left-identity lemma above. However, when ACL2 tries to construct this proof, it fails to apply ***-left-id** because it cannot relieve the hypothesis **(true-listp (reverse y))** of our version of ***-left-id**. This fact requires induction to prove and ACL2 does not use induction to relieve hypotheses during backchaining. Therefore the user must help ACL2 by proving

```
(defthm true-listp-reverse
  (true-listp (reverse x)))
```

first. Then ACL2 "automatically" proves

```
(defthm reverse-*
  (equal (reverse (* x y)) (* (reverse y) (reverse x)))).
```

McCarthy's $[reverse[reverse[x]] = x$ is not valid in our system; the restriction to lists ending in **nil** must be made explicit. However, the following is proved automatically

```
(defthm reverse-reverse
  (implies (true-listp x)
           (equal (reverse (reverse x)) x)))
```

and uses `reverse-*` in the proof. It is perhaps worth noting that `reverse-reverse` can also be proved by ACL2 immediately after the introduction of the definition of `reverse`, i.e., without any user-suggested lemmas about `reverse`. The prover's heuristics lead it to the conjecture `(equal (reverse (* x (cons a nil))) (cons a (reverse x)))`, instead of the more general `reverse-*`. It proves this conjecture by a second induction.

McCarthy's classic paper also deals with proofs in Peano arithmetic. These theorems can also be proved by ACL2.

ACL2's performance on problems of this scale is essentially the same as NQTHM's, which in turn is essentially the same as the Pure Lisp Theorem Prover of 1973. (It should be noted that the earliest version of our prover did not do termination proofs of function definitions and thus could be made inconsistent by user-added "definitional" axioms.)

6 Merge Sort

Here is a problem with which ACL2 has little difficulty but NQTHM has much more. The ACL2 solution exploits ACL2's support for equivalence relations and congruence based rewriting. We will define a merge sort algorithm and prove that it returns an ordered permutation of its input. The algorithm is defined as follows:

```
(defun merge-sort (x)
  (if (consp x)
      (if (consp (cdr x))
          (merge (merge-sort (evens x))
                 (merge-sort (odds x)))
        x)
    x))
```

where we will define `(evens x)` and `(odds x)` to return those elements of the linear list `x` in even-numbered or odd-numbered positions, respectively, and we will define `merge` so that it merges two ordered lists.

Our goal is to prove both `(orderedp (merge-sort x))` and `(perm (merge-sort x) x)`. The first establishes that the output of `merge-sort` is ordered. The second establishes that the output is a permutation of the input. With an ideal theorem prover we should only have to define the concepts involved and state the two theorems. With ACL2 we must do more.

6.1 Basic Concepts: Ordered Permutations

Here is the definition of

```
(defun orderedp (x)
  (if (consp x)
      (if (consp (cdr x))
          (and (<= (car x) (cadr x))
               (orderedp (cdr x)))
        t)
    t))
```

For example, `(orderedp '(1 2 2 7 13))` is `t` but `(orderedp '(1 2 7 2 13))` is `nil`. ACL2 admits this definition without help since its standard measure, `(acl2-count x)` decreases.

Below is the definition of when one list is a permutation of another. The basic idea is that two lists are permutations if and only if we can successively find and remove equal elements from the two lists until both lists are empty.

```
(defun perm (x y)
  (if (consp x)
      (and (mem (car x) y)
           (perm (cdr x) (del (car x) y)))
      (not (consp y))))
```

where `(mem e x)` determines whether `e` is a member of `x` and `(del e x)` deletes the first occurrence of `e` from `x`. Their easily admitted definitions are:

```
(defun mem (e x)
  (if (consp x)
      (or (equal e (car x))
          (mem e (cdr x)))
      nil))

(defun del (e x)
  (if (consp x)
      (if (equal e (car x))
          (cdr x)
          (cons (car x) (del e (cdr x))))
      nil))
```

`Perm` is admitted easily too, using `(acl2-count x)` as the measure. ACL2 can be made to prove that `perm` is an equivalence relation. That is

```
(defthm perm-is-an-equivalence
  (and (perm x x)
       (implies (perm x y) (perm y x))
       (implies (and (perm x y) (perm y z)) (perm x z)))
  :rule-classes :equivalence).
```

This may be abbreviated as `(defequiv perm)` using a macro that expands as above. We do not show the lemmas here. NQTHM can, of course, be made to prove this theorem, but NQTHM does not subsequently make any special use of it.

Once `perm` is known to be an equivalence relation, ACL2 can easily prove the following congruence rule for `cons`: `(implies (perm x y) (perm (cons e x) (cons e y)))`. This may be abbreviated as the command:

```
(defcong perm perm (cons x y) 2)
```

and can be thought of as telling ACL2 that if one is rewriting a `cons` expression while maintaining the equivalence relation `perm`, it is permitted to rewrite the second argument while maintaining `perm`. That is, theorems about `perm` may be used as rewrite rules in that context. ACL2 can also easily prove

```
(defcong perm perm (append x y) 1)
(defcong perm perm (append x y) 2)
```

which mean that when rewriting `append` expressions while maintaining the `perm` equivalence relation, it is permitted to rewrite both arguments maintaining `perm`. Finally, ACL2 can easily prove

```
(defthm append-cons
  (perm (append x (cons a y)) (cons a (append x y))))
```

which says that `(append x (cons a y))` is a permutation of `(cons a (append x y))`. This lemma is interpreted as a rewrite rule (in contexts in which we are maintaining the `perm` equivalence) directing ACL2 to replace any instance of `(append x (cons a y))` by the corresponding instance of `(cons a (append x y))`.

The definitions and lemmas mentioned in this section may be collected together into a "book" and incorporated into subsequent ACL2 sessions by the user. The ability to construct the database of relevant theorems incrementally is a very powerful feature of ACL2 because it allows users to build on the work of others and encourages the development of general collections of rules. We will call the book representing the work of this section `"orderedp-and-perm"`. In all it contains four definitions (`orderedp`, `mem`, `del`, and `perm`) and five theorems,

```
(defequiv perm)
(defcong perm perm (cons x y) 2)
(defcong perm perm (append x y) 1)
(defcong perm perm (append x y) 2)
(defthm append-cons
  (perm (append x (cons a y)) (cons a (append x y))))
```

The proof of `(defequiv perm)` requires the proof of seven lemmas which are not "exported" from the book. Henceforth we assume we have the material in the `"orderedp-and-perm"` book. We will now solve the merge sort problem while showing every lemma and hint the user must give to ACL2.

6.2 The Definition of Merge-Sort

We define the even and odd elements of a list as follows:

```
(defun evens (x)
  (if (consp x)
      (cons (car x) (evens (cddr x)))
    nil))

(defun odds (x) (evens (cdr x)))
```

Their admission is trivial. To merge two lists we define

```
(defun merge (x y)
  (declare (xargs :measure (+ (acl2-count x) (acl2-count y))))
  (if (consp x)
      (if (consp y)
          (if (< (car x) (car y))
              (cons (car x) (merge (cdr x) y))
            (cons (car y) (merge x (cdr y))))
        x)
    y))
```

Note that it is necessary to tell ACL2 a suitable measure.

Our next goal is to admit the previously shown definition of `merge-sort`. It recurs on the `evens` and `odds` of its argument, when that argument is a list of at least two elements. We must prove that the sizes of these two lists are smaller. That can be done with the following lemma.

```
(defthm acl2-count-evens
  (and (implies (and (consp x)
                     (consp (cdr x)))
                (< (acl2-count (evens x))
                   (acl2-count x)))
       (<= (acl2-count (evens x))
           (acl2-count x)))
  :rule-classes :linear)
```

This lemma, which is really two interesting mathematical facts, says that the size of **(evens x)** is weakly smaller than than of **x** and the inequality is strict when **x** has more than one element. These facts are proved without further ado. But the **defthm** command above has a third interesting aspect. It directs the system to build these facts into the linear arithmetic package. They are used automatically during the admission of **merge-sort** (above).

6.3 Proof of Merge-Sort's Correctness

The system is now able to prove

```
(defthm orderedp-merge-sort
  (orderedp (merge-sort x)))
```

without further help from the user. The system's heuristics lead it to conjecture that **(merge x y)** is ordered if **x** and **y** are ordered, and to prove this conjecture with a secondary induction. The reader is invited to construct the proof of the **merge** lemma by hand. Most readers who try will probably feel that their hand proof is messy and boring, with various cases having to be considered. The fact that ACL2 does the proof automatically illustrates McCarthy's hope that we can require the computer to do more work than a person is willing to check.

The last goal is that **merge-sort** produces a permutation of its input. We contribute to the proof in a major way by making two key observations. First, **merge** is **append** (modulo the **perm** sense of equivalence).

```
(defthm perm-merge
  (perm (merge x y) (append x y))

  :hints (("Goal" :induct (merge x y))))
```

Note that we have to tell ACL2 to induct the way **merge** recurses (instead of the way **append** recurses). Once proved, this theorem allows ACL2 to replace instances of **(merge x y)** by instances of **(append x y)** in contexts in which **perm** equivalence is to be maintained.

The second key observation is that appending the **evens** and **odds** of **x** produces **x** (modulo **perm**).

```
(defthm perm-evens-odds
  (perm (append (evens x) (evens (cdr x)))
        x))
```

This too is used as a rewrite rule in appropriate contexts.

With these two observations in the database, the final goal is trivial by induction on **x**.

```
(defthm perm-merge-sort
  (perm (merge-sort x) x))
```

So how much extra work did we have to do? Let us assume we know the facts in `"orderedp-and-perm"`. This is not unreasonable since they are independent of the particular sorting algorithm we are analyzing.

The ideal prover would have required us to make four definitions (`evens`, `odds`, `merge` and `merge-sort`) and then state two theorems `orderedp-merge-sort` and `perm-merge-sort`. ACL2 required, in addition, that we

- exhibit the measure that decreases in `merge`,
- observe that the size of `evens` decreases, sometimes strictly,
- recommend that these facts be used as linear rules,
- observe that `merge` is `append` (modulo `perm`), and
- observe that `append` of the `evens` and `odds` is the identity (modulo `perm`).

We do not find these requirements onerous. The first three could probably be lifted with some additional heuristics. The last two seem to require genuine mathematical insight. At least ACL2 allows the user to communicate the insight and can make appropriate use of it. Of course, this aspect of ACL2 should remind readers of McCarthy's "Advice Taker" [21] proposal in which he says "In order for a program to be capable of learning something it must first be capable of being told it."

7 Computing Machines

ACL2 is most frequently used to model microprocessors and other computing machines. In this section we sketch how this is done.

7.1 Java Byte Code

Here is a Java program "implementing" the factorial function.

```
public static int fact(int n){
  if (n>0)
    {return n*fact(n-1);}
  else return 1;
}
```

To be precise, it implements the factorial function for values of `n` between 0 and 12. 13! is 6227020800 but the program above returns 1932053504 because the `int` type in Java is allocated only 32 bits. We will ignore this and other important fine points of the semantics of Java in this discussion but will return to it later.

Java is implemented by compiling it into "byte code" for the Java Virtual Machine or JVM [19]. The JVM is a stack based machine. In this section we show a formal model of a "toy JVM", TJVM, that is suggestive of the JVM but small enough to present in some detail. Below we show the compilation of the `fact` program above. In the left column is the byte code for our TJVM. On the right is the JVM code generated by Sun Microsystems' Java compiler.

```
("fact" (n)
   (load n)                         ;  0 iload_0
   (ifle 8)                         ;  1 ifle 12
   (load n)                         ;  2 iload_0
   (load n)                         ;  3 iload_0
   (push 1)                         ;  4 iconst_1
```

```
  (sub)                              ;  5 isub
  (invokestatic "Math" "fact" 1) ;  6 invokestatic <Method int fact(int)>
  (mul)                              ;  7 imul
  (xreturn)                          ;  8 ireturn
  (push 1)                           ;  9 iconst_1
  (xreturn))                         ; 10 ireturn
```

The left-hand column is actually a list constant containing thirteen elements. The first element is the string `"fact"`; the second is a list containing the single symbol `n`, the third is a list containing the symbol `load` followed by the symbol `n`, etc. This constant will be our representation of a "method declaration" on the TJVM. This method declaration for the `"fact"` method might be found in the `"Math"` class on the TJVM. The shallow correspondence between our TJVM and the JVM is obvious. Some of the discrepancies are explained as follows. On the TJVM, methods refer to their local variables by name; on the JVM methods refer to their locals by position. The "8" in the TJVM `ifle` instruction is an instruction offset by which the program counter is incremented. The corresponding offset on the JVM counts bytes rather than instructions and some JVM instructions take more than one byte. The JVM has typed instructions, e.g., the JVM's `iload` loads an integer-valued variable on the stack while the TJVM's `load` loads any value.

Roughly speaking, `(load n)` pushes the value of `n` on the stack; `(ifle 8)` pops one item off the stack and increments the program counter by `8`, if the item is less than or equal to 0, and by 1, otherwise; `(push 1)` pushes the constant `1`; `(sub)` pops two items off the stack and pushes their difference; and `(invokestatic "Math" "fact" 1)` invokes the method named `"fact"` in the `"Math"` class on the topmost object on the stack.

A mechanized theory of computation must allow us to formalize such machines and reason about their behavior mechanically.

7.2 The Formal Model

We do this by formalizing the notion of the state of the TJVM. Our TJVM state contains several components, including a call stack of frames, a heap, and a class table listing classes and their method declarations. Here, a frame is itself a structure associated with each method invocation. A frame contains the current program counter, the stack of intermediate results, the invocation's variable bindings, and the byte code of the invoked method. We then define the semantics of each TJVM instruction as a function which takes the instruction, `inst`, and the current TJVM state, `s`, and returns a new state. Below is the semantic function for the TJVM `add` instruction. We use a macro named `modify` to denote the state obtained from `s` by replacing the indicated components and leaving all other components unchanged.

```
(defun execute-add (inst s)
  (modify s
          :pc (+ 1 (pc (top-frame s)))
          :stack (push (+ (top (pop (stack (top-frame s))))
                          (top (stack (top-frame s))))
                       (pop (pop (stack (top-frame s)))))))
```

The `add` instruction increments the program counter (in the topmost frame of the call stack of `s`) and pops two items off the stack (in the topmost frame of the call stack of `s`) and pushes their sum. In this model, the sum is computed with the ACL2 function `+`, which is true mathematical addition, not the 32-bit 2's complement sum operator used by the JVM.

After defining the semantic function of each TJVM byte code instruction we define the "big switch" `do-inst` which executes a given instruction on a state, we use it to define `step`, which executes the next instruction (i.e., the one at the program counter on the topmost frame of the call stack), on the current state, and finally define `tjvm`, which iteratively steps the current state `n` times.

```
(defun do-inst (inst s)
  (case (op-code inst)
    (add            (execute-add inst s))
    ...
    (invokevirtual  (execute-invokevirtual inst s))
    (invokestatic   (execute-invokestatic inst s))
    (getfield       (execute-getfield inst s))
    (putfield       (execute-putfield inst s))
    ...
    (otherwise s)))

(defun step (s)
  (do-inst (next-inst s) s))

(defun tjvm (s n)
  (if (zp n)
      s
    (tjvm (step s) (- n 1))))
```

The function **tjvm** is an ACL2 version of a state machine. This machine is not finite, since we can use it to create arbitrary integers by repeated **add**s.

7.3 Theorems about TJVM

As befitting a mechanized theory of computation, we can prove theorems mechanically about such state machines. A very useful general theorem about **tjvm** is the following:

```
(defthm tjvm-+
  (implies (and (natp i) (natp j))
           (equal (tjvm s (+ i j))
                  (tjvm (tjvm s i) j))))
```

which allows us to decompose a long run into two shorter runs. This theorem is proved automatically by induction on **i**. The theorem is quite general, putting no restrictions on (the state) **s**.

At the opposite extreme is a theorem that can be proved simply by execution. Consider the state to which **tjvm** below is applied. It is a constant: a state with one frame, poised to push **13** on the empty stack and call our **"fact"**. Suppose **tjvm** is used to take **124** steps from that state and then consider the topmost item on the stack in the topmost frame of the call stack of the resulting state. The theorem says we will find **6227020800**. Here the constant **(Math-class)** is defined to be a class declaration containing the **"fact"** method declaration shown earlier.

```
(defthm example13
  (equal (top
          (stack
           (top-frame
            (tjvm (make-state (push (make-frame 0 nil nil
                                                '((push 13)
                                                  (invokestatic "Math" "fact" 1)
                                                  (halt)))
                                    nil)
                              nil (list (Math-class)))
                  124))))
         6227020800))
```

This illustrates the executability of ACL2. It is easy to see that this might be a useful feature when dealing with situations like this. We use test executions to experiment with our formal models. An important aspect of this experimentation is to *corroborate* a formal model against an informal "specification" such as the expectations of the design team, a document written in English or some other informal language, or the behavior of some artifact such as a simulator or prototype created by the designers. In addition, if an expression such as that above arises during the course of a more difficult proof, it is convenient to be able to compute its value, just as one might wish to compute, rather than derive by primitive proof steps, the values of `(+ 2 2)` or `(expt 2 32)`.

Finally, here is another theorem we can prove about `tjvm`. It states that if `s0` is a TJVM state poised to execute a call of our `"fact"` method on a natural number `n`, then the state obtained by executing `s0` a certain number of steps, namely `(fact-clock n)`, can alternatively be obtained by advancing the program counter of `s0` by one and pushing `(fact n)` in place of `n` on the stack.

```
(defthm fact-is-correct
  (implies (and (equal (next-inst s0) '(invokestatic "Math" "fact" 1))
                (equal (assoc-equal "Math" (class-table s0))
                       (Math-class))
                (equal n (top (stack (top-frame s0))))
                (natp n))
           (equal
            (tjvm s0 (fact-clock n))
            (modify s0
                    :pc (+ 1 (pc (top-frame s0)))
                    :stack (push (fact n) (pop (stack (top-frame s0))))))))
```

Here, `fact` is defined as

```
(defun fact (n)
  (if (zp n)
      1
    (* n (fact (- n 1)))))
```

The theorem above establishes that the TJVM byte code for our `"fact"` method is correct in the sense that it computes the same thing as the factorial function in ACL2. Furthermore, as a rewrite rule it is extremely useful because it allows us to run `tjvm` past an invocation of `"fact"` in a single logical step, just as though `(invokestatic "Math" "fact" 1)` were a primitive instruction that computed the factorial of the top of the stack, except costing `(fact-clock n)` primitive steps.

For details about the TJVM model, see [26]. For details about how ACL2 is configured to prove such theorems, see [5].

8 Nondeterminism and Distributed Computation

Within this basic framework it is possible to formalize machines that are nondeterministic by incorporating into the model an "oracle" that provides "random" input for each step. An example of this, as well as of the modeling of distributed computation, is discussed in [25].

A model is given of a set of n machines, each more or less comparable to the TJVM, but operating on a shared memory via a "compare and swap" instruction. Parallelism is modeled by interleaved atomic execution of individual instructions. An oracle determines which processor steps next. To illustrate the efficacy of the model, we prove a theorem about the execution of a certain non-blocking counter algorithm. The details of the algorithm are unimportant here.

But the proof involved formalizing a different view of the system: that of a single processor in the system. From that perspective, only one processor exists, but the shared part of memory changes "chaotically." Formalizing this required another oracle.

The proof strategy then involved proving an invariant about the shared memory in the multiprocessor system, translating that invariant to a restriction on the oracle in the uniprocessor view, and then doing a classical uniprocessor code proof (such as that for `"fact"` above) under the restricted oracle. Of course, the two models of computation have to be formally related if conclusions reached in the uniprocessor view are to be reflected in the multiprocessor view in which the goal theorem was stated.

Relating the two models involved showing how a computation on the multiprocessor system could be carried out on the uniprocessor system from the perspective of any selected processor, **pid**. In a logic providing existential quantification, the required theorem simply states that for all states and oracles of the multiprocessor system there exist a state and oracle of the uniprocessor system such that the two systems yield "the same" answers on the respective states and oracles. (The notion of equivalence is not equality since the notions of "state" in the two systems are different.) But ACL2 does not have existential quantification. Instead of claiming something exists we must construct the alleged object. That is, we must define functions that map a state and oracle of one system to a state and oracle of the other. This is not as distracting as it may seem because, generally, when one proves an existential theorem the proof encodes such a construction.

We focus here on the transformation of the oracle. The goal is to produce from a multiprocessor state and oracle, the sequence of memories seen each time a selected process, **pid**, is stepped by the uniprocess model. We first partition the oracle of the multiprocessor system, slicing it at each clock tick at which **pid** is stepped. The multiprocessor system was then run on each successive partition, starting each run in the appropriate multiprocessor state. The resulting shared memories are then collected as the uniprocessor oracle. This rather messy and ambiguous English description can be succinctly written in ACL2 as:

```
(defun new-oracle (pid s partitions)
  (if (endp partitions)
      nil
      (cons (mem (mrun s (car partitions)))
            (new-oracle pid
                        (mstep pid (mrun s (car partitions)))
                        (cdr partitions)))))
```

where **partitions** is the multiprocessor oracle after slicing it at each **pid**, and **mstep** and **mrun** are the multiprocessor single- and iterated step transformations.

The modeling methods sketched here can be extended to deal with much more complex systems. For example, Sawada [31] has constructed an ACL2 model of a pipelined microprocessor with multiple instruction issue and speculative execution. His model includes interrupts and exceptions. He uses ACL2 to prove that the modeled processor implements a sequential instruction set architecture. The combination of the pipeline and external interrupts makes this an especially difficult task, requiring very clear thinking about "when" an interrupt is "seen" by the higher level machine. It is unlikely that a suitable explanation would be given without the aid of mechanized proof.

There are other ways of dealing with nondeterminism in ACL2, including the use of constrained functions or the definition of state machines in terms of a state transition relation rather than a step function.

In [20], Manolios, Namjoshi and Sumners show a method of using ACL2 to prove that two state machines are bisimilar up to stuttering. The method, based on well-founded orderings, allows the proof to focus on single state transitions rather than on infinite execution paths. They demonstrate their method by verifying the alternating bit protocol. The protocol, as initially defined, has an infinite state space. But it is shown stuttering bisimilar to a small finite-state system, which is then model-checked.

9 Industrial Applications

ACL2 finds its most important applications in industry, where it is used to prove properties of models of hardware and low-level software.

For example, ACL2 has been used to prove theorems about formal models of

- the Motorola CAP digital signal processor [7],
- the floating-point units of the AMD K5 and K7 microprocessors [27, 30].
- the Rockwell-Collins JEM1 microprocessor (the world's first silicon Java Virtual Machine) [13, 14], and
- the IBM 4758 secure coprocessor [33],

With the exception of the IBM 4758 (which involved a security model) the models concerned are executable and are both bit- and cycle-accurate. That is, the models completely specify the value of every bit in the machine, on every cycle. Thus, these models are extremely large and complex. Often, however, they follow very closely the style of formalization shown in the TJVM examples above.

In the Motorola CAP work, ACL2 was used to prove the correspondence between two such models. One modeled the pipelined microarchitectural design of a digital signal processor. The other modeled the sequential instruction set level. The theorem proved by Bishop Brock [7] established that the two models were "equivalent" in a suitable sense, provided that the microcode being executed did not expose certain pipeline "hazards." A predicate was defined to capture this notion of whether microcode contained pipeline hazards. One can view the major contribution of this work as being the identification and certification of this predicate. The theorem just cited can be thought of a correctness result for the predicate: the designers' model of the hardware and the programmers' model are equivalent for code satisfying the predicate. The predicate is executable. So it was possible to ascertain, without further proof, whether microcode written for the CAP was hazard free. To this end, Brock extracted from the CAP ROM about 50 digital signal processing applications, written by Motorola microcode programmers, and simply executed his predicate on the microcode. Several hazard violations were found; they were eventually recognized as errors and fixed. Whether these errors would have been found by conventional testing is, of course, unknown. But these kinds of errors can be difficult to find: the low level machine is working as the designers expect, the microcode is correctly written for the high level machine, and the problem is that the low level machine does not necessarily implement the high level machine because of the pipeline hazards.

In related work, similar to the proof of the TJVM `"fact"` program, Brock and Moore proved that microcode for a systolic peak finding algorithm was correct [8].

The initial AMD work proved that the microcode for the AMD K5 correctly implements floating point division according to the IEEE standard [28]. That standard says that the result computed shall be the floating point number that would be obtained by computing the actual quotient to infinite precision and then rounding it according to the rounding mode specified by the user. The microcode was formalized by hand-translating it to an ACL2 function. Here is a sketch of the function:

```
(defun divide (p d mode)
 (let* ((sd0    (eround (lookup d)                 '(exact 17 8)))
        (dr     (eround d                          '(away 17 32)))
        (sdd0   (eround (* sd0 dr)                 '(away 17 32)))
        (sd1    (eround (* sd0 (comp sdd0 32))     '(trunc 17 32)))
        (sdd1   (eround (* sd1 dr)                 '(away 17 32)))
        (sd2    (eround (* sd1 (comp sdd1 32))     '(trunc 17 32)))
        ...
        (qq2    (eround (+ q2 q3)                  '(sticky 17 64)))
```

```
  (qq1    (eround (+ qq2 q1)                  '(sticky 17 64)))
  (divide (round   (+ qq1 q0)                  mode)))
 (or (first-error sd0 dr sdd0 sd1 sdd1 sd2 dh dl p0 ph0 ... divide)
     divide)))
```

The function takes three inputs, `p`, `d`, and `mode`. The first two are floating-point numbers and the last is the desired rounding mode for the quotient. Roughly speaking, the function sequentially computes values for the variables **`sd0`**, **`dr`**, **`sdd0`**, etc., and then either returns the first error that was detected or the value of the last variable, **`divide`**. Errors are modeled by non-numeric values, like strings, and we prove that no error occurs. The variables here represent floating-point registers in the microcode. Consider the line above at which **`sdd1`** is given a value. Its value is the (infinitely precise) product of **`sd1`** and **`dr`**, rounded according to the mode **`(away 17 32)`**, which is to say, to 32 bits of precision and 17 bits of exponent. The **`eround`** function signals an error if the product is not representable in 17 bits of exponent. Given the IEEE standard, this line is just the formal way of saying that **`sdd1`** is the IEEE product of **`sd1`** and **`dr`**, rounding according to **`(away 17 32)`** and represents one line of the microcode. To formalize **`divide`**, above, required formalizing that part of the IEEE standard that deals with elementary operations and rounding.

ACL2 is executable. For example, **`(divide 1 3 '(trunc 15 48 mask-uv))`** divides `1` by `3` and truncates it to 48 bits of precision and 15 bits of exponent. The result computed by ACL2 is the rational `93824992236885/281474976710656`. The binary representation of this rational is `0.0101010101010101010-101010101010101010101010101`. This is the same result given by AMD's simulator (and by the K5 after it was fabricated).

The following theorem was proved about **`divide`**.

```
(defthm divide-divides
  (implies (and (floating-point-numberp p 15 64)
                (floating-point-numberp d 15 64)
                (not (equal d 0))
                (rounding-modep mode))
           (equal (divide p d mode)
                  (round  (/ p d) mode))))
```

Less formally, if p and d are floating point numbers, $d \neq 0$, and *mode* is a rounding mode, then the **`divide`** function shown above produces the result of rounding the infinitely precise p/d to a floating point number as specified by *mode*. This work was done in collaboration with Matt Kaufmann and Tom Lynch, the leader of the AMD K5 floating point design team [27].

To lead ACL2 to this theorem required proving about 1000 lemmas. Many of them are general facts about the relationships between the elementary operations and the various rounding styles.

ACL2 work at AMD continues on the as-yet-to-be-released AMD K7 project. David Russinoff has used ACL2 to analyze the hardware designs for parts of the AMD K7. He produced a mechanical translator from the RTL design language used at AMD to ACL2 and then verified mechanically the correctness of the K7 hardware designs for the floating-point operations of addition, subtraction, multiplication, division, and square root. He corroborated his translation from RTL to ACL2 by running the ACL2 model of square root on 80 million test vectors, getting precisely the same results as AMD's RTL simulator. While attempting to construct and mechanically check correctness proofs he found several bugs in these well-tested designs. The bugs were fixed by changes to the RTL designs and the new designs were mechanically verified. See [30]. One can only speculate as to whether the bugs would have been found by traditional testing before the K7 was fabricated and shipped. The first Intel Pentium chip contained a bug in the floating point operation and Intel is reputed to have spent $500 million fixing it after the chip was shipped to customers.

Researchers at Rockwell-Collins Avionics have produced the world's first silicon Java Virtual Machine and used ACL2 to model it formally. Unlike our TJVM, this model of the JVM is intended to be faithful to the actual design. Proofs were done on some minor parts of the design, primarily to demonstrate that proofs were possible. The main thrust of the work was to produce a formal model that was an efficient execution engine. The ACL2 model can execute about 250,000 JEM1 instructions per second, exploiting ACL2 single-threaded objects [4]. With some behind-the-scenes work, the Rockwell engineers have improved this to over 2 million [13, 14]. Their model is integrated into a simulation environment so that other Rockwell engineers use it to run compiled Java byte code without knowing they are using a formal model.

In related work, Rich Cohen has formalized (a subset of) Sun's informal specification of the JVM. Cohen's model, called the "defensive JVM" or dJVM [9] signals errors which are supposed to be avoided in the actual JVM by virtue of the checks made by Sun's byte code verifier. It is, in principle, possible to use the dJVM to investigate whether the verifier successfully guards against these errors.

Matt Kaufmann has used ACL2 to verify that certain transformations on COBOL date processing code produces Y2K compliant results. This work was done in support of commercial Y2K "remediation tools" with which programmers at EDS, Inc., attempt to find and correct COBOL code that makes now outdated assumptions about the century. Kaufmann's basic idea was to formalize the actual date represented by certain common data types, e.g., six-digit year-month-day format, and prove that certain extractions and comparisons produced results consistent with the intended interpretation of "earlier," "difference", etc. See [15].

An emerging application of formal methods is e-commerce, an area where security and reliability are of great importance. At IBM Yorktown, ACL2 was used for the FIPS level 4 evaluation of the IBM 4758 secure coprocessor [33]. This was the first device ever to achieve the highest certification level (4) for cryptographic devices, according to Federal Information Processing Standards (FIPS) 140-1; it is (to date) the only device to have achieved level 4 for both hardware and software. The 4758, about the size of a book, can be used to store confidential data unguarded in a hostile environment. This and other characteristics make the 4758 especially attractive for e-commerce applications. The ACL2 work involved formalizing a "security model" for the IBM 4758 and proving certain high level theorems about that model.

10 Conclusions

ACL2 is not the general-purpose theory of computation envisioned by McCarthy. The logic supported by ACL2 is much weaker than that used by McCarthy. For example, he freely uses quantifiers, higher order functions and set theory and can thus investigate questions that fall outside the realm of ACL2.

Why is ACL2 so weak? It is certainly possible to construct mechanical proof checkers for more expressive logics [12, 29] and these systems are used for applications similar to ours. But two good arguments can be made for ACL2's expressive weakness. The first is that ACL2 was designed with mechanical proof in mind and hence represents a compromise between the two extremes of propositional calculus – where little of interest can be said but everything valid can be proved automatically – and heavy duty set theory – where everything of interest can be said but little can be proved automatically. The second argument is that executability is important for the applications we have in mind. Indeed, the main reason I abandoned almost twenty years of work on the NQTHM logic and started with ACL2 was the conviction that to construct and corroborate models of realistic microprocessors it was necessary to make the logic execute much more efficiently.

As illustrated here, the ACL2 logic is applicable to a wide variety of problems, despite its expressive weakness. This may account for its appeal: it is simple but often just powerful enough to get the job done. In addition, no matter what heavy duty features a logic may have, it will be necessary to reason with grace and power about elementary recursive functions of classic data structures like integers, lists and trees. Hence, I personally see the NQTHM and ACL2 projects as carrying out an exploration that is essential to the eventual success of formal methods, regardless of the power of the logical systems eventually adopted.

11 Acknowlegements

I am indebted to Bob Boyer and Matt Kaufmann for their years of collaboration on both NQTHM and ACL2. In addition, I would like to thank the many ACL2 users whose work has been cited here, including Vernon Austel, Bishop Brock, Rich Cohen, John Cowles, Art Flatau, David Greve, David Harding, Warren Hunt, Tom Lynch, Pete Manolios, Kedar Namjoshi, David Russinoff, Jun Sawada, Rob Sumners, and Matt Wilding. I am also indebted to the other ACL2 and NQTHM users who, for almost three decades, have fearlessly applied the systems to problems well beyond those for which they were designed and in so doing showed us what is necessary in a mechanized theory of computation. No project as ambitious as the construction and adoption of a mechanized theory of computation can be carried out without the enthusiastic support and hard work of such risk-taking innovators.

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, Second Edition*. Academic Press, New York, 1997.

[4] R. S. Boyer and J S. Moore. Single-threaded objects in acl2. *(submitted for publication)*, 1999.

[5] R. S. Boyer and J S.Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 147–176. MIT Press, 1996.

[6] 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.

[7] B. Brock, M. Kaufmann, and J S. Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag, LNCS 1166, November 1996.

[8] B. Brock and J S. Moore. A mechanically checked proof of a comparator sort algorithm. *(submitted for publication)*, 1999.

[9] R. M. Cohen. The defensive java virtual machine specification, version 0.53. Technical report, Electronic Data Systems Corp, Austin Technical Services Center, 98 San Jacinto Blvd, Suite 500, Austin, TX 78701, 1997.

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

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

[12] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[13] D. A. Greve. Symbolic simulation of the JEM1 microprocessor. Technical Report (submitted), Advanced Technology Center, Rockwell Collins, Inc., Cedar Rapids, IA 52498, 1998.

[14] D. A. Greve and M. M. Wilding. Stack-based Java a back-to-future step. *Electronic Engineering Times*, page 92, Jan. 12, 1998.

[15] M. Kaufmann. Acl2 support for verification projects. In C. Kirchner and H. Kirchner, editors, *Proceedings 15th International Conference on Automated Deduction*, pages 220–238. LNAI 1421, Springer-Verlag, July 1998.

[16] M. Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[17] M. Kaufmann and J S. Moore. *The ACL2 User's Manual.* http://www.cs.utexas.edu/users/moore/-acl2/acl2-doc.html#Users's-Manual, 1999.

[18] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *(submitted for publication)*, 1999.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[20] P. Manolios, K. Namjoshi, and R. Sumners. Linking theorem proving and model-checking with well-founded bisimulation. In *Computed Aided Verification: 10th International Conference, CAV '99.* Springer-Verlag LNCS, 1999.

[21] J. McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes.* H.M.Stationery Office, 1959.

[22] J. McCarthy. Towards a mathematical science of computation. In *Proceedings of IFIP Congress*, pages 21–28. North-Holland, 1962.

[23] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems.* North-Holland Publishing Company, Amsterdam, The Netherlands, 1963.

[24] J S. Moore. Introduction to the obdd algorithm for the atp community. *Journal of Automated Reasoning*, 12(1):33–45, 1994.

[25] J S. Moore. A mechanically checked proof of a multiprocessor result via a uniprocessor view. *Formal Methods in System Design*, 14(2):213–228, March 1999.

[26] J S. Moore. Proving theorems about java-like byte code. In E.-R. Olderog and B. Steffen, editors, *Correct System Design – Issues, Methods and Perspectives.* LNCS, to appear, 1999.

[27] J S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm. *IEEE Transactions on Computers*, 47(9):913–926, Sep 1998.

[28] Standards Committee of the IEEE Computer Society. Ieee standard for binary floating-point arithmetic. Technical Report IEEE Std. 754-1985, IEEE, 345 East 47th Street, New York, NY 10017, 1985.

[29] 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. Lecture Notes in Artificial Intelligence, Vol 607, Springer-Verlag, June 1992.

[30] D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division, and square root algorithms of the AMD-K7 processor. *http://www.onr.com/user/russ/david/k7-div-sqrt.html*, 1998.

[31] J. Sawada and W. Hunt. Processor verification with precise exceptions and speculative execution. In *Computed Aided Verification: 10th International Conference, CAV '98*, pages 135–146. Springer-Verlag LNCS 1427, 1998.

[32] J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Ma., 1967.

[33] S. W. Smith and V. Austel. Trusting trusted hardware: Towards a formal model for programmable secure coprocessors. In *Proceedings of the Thrid USENIX Workshop on Electronic Commerce*, September 1998.